CARL VON OSSIETZKY universität OLDENBURG

Studiengang Informatik

Diplomarbeit

Trace- und Zeit- Zusicherungen beim Programmieren mit Vertrag

vorgelegt von

Mark Brörkens

Betreuender Gutachter	Zweiter Gutachter
Prof. Dr. Ernst-Rüdiger Olderog	Dipl.-Inform. Dietrich Boles

Oldenburg, 31. Januar 2002

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Die Entwicklungen der Computertechnik breiten sich immer stärker im Bereich unseres täglichen Lebens aus. Dabei werden Software-Systeme zunehmend für sicherheitskritische Aufgaben eingesetzt: Sie steuern Züge, Flugzeuge und Raketen, kontrollieren Kraftwerke und medizinische Geräte. Fehler in der Software können Menschenleben kosten. Trotz moderner Methoden bei der Softwareentwicklung enthalten fast alle Programme Fehler.

In den letzten zwanzig Jahren wurde viel an Methoden zur Analyse und Verifikation von Software-Systemen geforscht. Viele Fallstudien aus dem industriellen Umfeld zeigen die Anwendbarkeit formaler Methoden (Clarke, Wing, Alur, et al. 1996). Die vollständige formale Verifikation ist allerdings häufig nicht möglich. Einerseits überschreitet die Größe und Komplexität vieler Systeme die Möglichkeiten formaler Verifikationstechniken. Andererseits beziehen sich Verifikations-Methoden meist auf formale Modelle des Systems und nicht auf die konkrete Implementierung selbst. Selbst wenn das Modell erfolgreich formal verifiziert ist, kann die Implementierung Fehler enthalten. Dies resultiert aus der meist höheren Detailliertheit der Implementierung oder aus geringen Abweichungen der Implementierung von der formalen Spezifikation. (Kim, Kannan, Lee, Sokolsky, und Viswanathan 2001)

Meyer (1998) zeigt mit dem Konzept des *Programmierens mit Vertrag* eine Möglichkeit auf, Elemente einer formalen Spezifikation direkt in die Implementierung aufzunehmen. Zur Laufzeit des Programmes wird die Einhaltung der Spezifikation geprüft. Verhält sich das Programm nicht entsprechend der Spezifikation, wird dies angezeigt. Obwohl das Programm auf diese Weise nicht

vollständig verifiziert werden kann, ermöglicht es aber, einzelne Programmabläufe zu validieren und somit die Zuverlässigkeit der Software zu erhöhen.

Heutige Software-Systeme sind meist aus mehreren Komponenten aufgebaut. Für die korrekte Funktionsweise der Komponenten miteinander sind häufig bestimmte Abläufe einzuhalten. Bevor zum Beispiel eine Anfrage an eine Datenbank gestellt werden kann, muss eine Verbindung zu ihr aufgebaut werden.

In der Arbeit von Plath (2000) wird das Konzept des *Programmierens mit Vertrag* um Trace-Zusicherungen erweitert. Eine Trace (zu deutsch: Spur) beschreibt eine Abfolge von Methoden-Aufrufen eines Programmes. Die Trace-Zusicherungen spezifizieren, in welcher Reihenfolge einzelne Methoden aufgerufen werden dürfen. Diese Art von Zusicherungen wird besonders bei größeren komponentenbasierten Applikationen interessant.

Im Umfeld eingebetteter Systeme ist häufig die Einhaltung von Zeitschranken wichtig. Ein System, das beispielsweise im Falle eines Unfalls den Airbag im PKW auslöst, ist unbrauchbar, wenn sich der Airbag beim Aufprall zu spät öffnet. Bei der Entwicklung derartiger Systeme wäre ein Tools zur Kontrolle des zeitlichen Verhaltens hilfreich.

Im Rahmen dieser Arbeit wird ein Tool entwickelt, das zur Laufzeit eines Java-Programmes Trace- und Zeit-Zusicherungen prüft.

1.2 Zielsetzung

Der Schwerpunkt der Arbeit liegt im Entwurf und der Implementierung eines Tools zur Prüfung von Trace- und Zeit-Zusicherungen zur Laufzeit eines Java-Programmes. Dafür werden die vier folgenden Aufgabenbereiche bearbeitet.

Analyse vorhandener Ansätze und Tools

In den letzten Jahren wurden Ansätze und Tools entwickelt, die die Programmiersprache Java um das Konzept des *Programmierens mit Vertrag* erweitern. Die verwendeten Techniken zur Integration des Konzeptes in Java werden miteinander verglichen. Die Ergebnisse dienen als Grundlage für den Entwurf des Tools.

Erläuterung technischer Grundlagen

Das Tool wird in der Programmiersprache Java implementiert. Die zur Umsetzung notwendigen technischen Grundlagen werden beschrieben.

Formale Definition einer Sprache zur Beschreibung von Traces

Die Trace- und Zeit-Zusicherungen werden in einem Dialekt der Sprache CSP (”Communicating Sequential Processes”) beschrieben. Die Syntax und Semantik der Sprache wird formal eingeführt.

Implementierung des Tools

Ein Prototyp des Tools wird implementiert. Dieser setzt die vorher erarbeiteten Konzepte um. Die Funktionalität wird anhand einfacher Beispiele überprüft. Um Aussagen über der Validierbarkeit von Zeit-Zusicherungen in Java-Programmen zu erhalten, wird der Einfluss des Tools auf das Laufzeitverhalten des untersuchten Programmes analysiert.

1.3 Aufbau der Arbeit

Im Kapitel 2 wird das Konzept des *Programmierens mit Vertrag* vorgestellt. Vorhandene Ansätze und Tools zur Erweiterung der Programmiersprache Java um dieses Konzept werden beschrieben und verglichen. Die Ergebnisse legen den Grundstein für die **J**ava with **Ass**ertions **D**ebugger **A**rchitecture (kurz: Jassda), einer Architektur zur Prüfung von Zusicherung zur Laufzeit von Java-Programmen

Kapitel 3 erläutert die technischen und theoretischen Grundlagen für den Entwurf und die Implementierung der Architektur. Es werden sowohl grundlegende Techniken des Debuggens erläutert als auch die Java Platform Debugger Architecture (JPDA) und das Java Debug Interface (JDI) beschrieben. Der zweite Teil des Kapitels beschäftigt sich mit der formalen Spezifikation von Trace- und Zeit-Zusicherungen.

Das Kapitel 4 geht genauer auf die *Java with Assertions Debugger Architecture* ein und beschreibt deren Implementierung. Es wird unter anderen beschrieben, wie Jassda eine Trace eines Programmes ermittelt und diese gegen eine Spezifikation prüft.

Die Fallstudie in Kapitel 5 untersucht, welchen zeitlichen Einfluss das Jassda-Tool auf das geprüfte Programm hat. Außerdem wird die Anwendbarkeit von Jassda für unterschiedliche Java-Applikationen nachgewiesen.

Das Kapitel 6 fasst die Ergebnisse dieser Arbeit zusammen und gibt einen Ausblick auf Verbesserungen und Erweiterungen.

1.4 Vorbemerkungen

Viele Fachbegriffe in der Informatik sind in englischer Sprache. Deutsche Übersetzungen sind häufig nicht vorhanden oder verzerren die Aussage. In dieser Arbeit wird daher auf die Übersetzung der Fachbegriffe verzichtet. Die Beispiele und Abbildungen als auch die Implementierung des Tools sind in englischer Sprache gehalten.
Bei der Recherche wurden viele interessante und nützliche Internetseiten und Java-Bibliotheken gefunden. Nicht alle konnten in dieser Arbeit aufgeführt werden. Die Internetseite `http://www.informatik.uni-oldenburg.de/~teddy/jassda` gibt einen Überblick der gefundenen Informationen sowie weitere Dokumentation zur Implementierung von Jassda.

Danksagung

Ich möchte mich bei allen Personen bedanken, die eine Realisierung dieser Arbeit möglich gemacht haben. Insbesondere danke ich der Firma Atanion, die mir in ihren Räumlichkeiten einen komfortablen Arbeitsplatz zur Verfügung gestellt hat. Detlef, Frank und Markus unterstützen mich durch unermüdliches Korrekturlesen, Udo konnte mir stets bei Latex-Problemen weiterhelfen, Mareike gab Hilfestellung bei der Erstellung der Abbildungen. Herzlichen Dank.
Herr Professor Dr. Olderog und Herr Dipl.-Inform. Möller standen mir während der Arbeit mit einer Fülle von Anregungen und Tipps zur Seite. Herzlichen Dank für die sehr gute Betreuung.

Kapitel 2

Programmieren mit Vertrag

Dieses Kapitel stellt das Konzept des *Programmierens mit Vertrag* vor und erläutert Ansätze und Tools, die die Programmiersprache Java um dieses Konzept erweitern. Die Vor- und Nachteile der einzelnen Ansätze und Tools werden aufgezeigt und zusammengefasst. Schließlich wird das im Rahmen dieser Diplomarbeit entwickelte Tool zur Validierung von Trace- und Zeitzusicherungen zur Laufzeit vorgestellt.

2.1 Einleitung

Bertrand Meyer entwickelte das Konzept des *Programmierens mit Vertrag* (englisch: Design by Contract) als eine zentrale Komponente der Programmiersprache Eiffel. *Programmieren mit Vertrag* ist jedoch auch auf andere Programmiersprachen übertragbar. (Meyer 2001)

2.1.1 Zusicherungen

Zusicherungen stellen einen grundlegenden Bestandteil des Konzepts des *Programmierens mit Vertrag* dar. Es handelt sich dabei um boolsche Ausdrücke, die bei einem korrekten Ablauf eines Programmes immer den Wert *wahr* ergeben. Sie werden typischerweise nur in der Entwicklungsphase eines Programmes überprüft und helfen speziell bei komplexen Systemen, Fehler frühzeitig zu erkennen. In der endgültigen Implementierung werden die Zusicherungen nicht mehr überprüft. Beim *Programmieren mit Vertrag* wird zwischen drei Typen von Zusicherungen unterschieden: Vor- und Nachbedigungen sowie Invarianten.

2.1.1.1 Vorbedingungen

Vorbedingungen spezifizieren Bedingungen, die vor der Ausführung einer Operation (vor dem Aufruf einer Java-Methode) gültig sein müssen. Beispielsweise sei eine Methode gegeben, die den Flächeninhalt eines Quadrates berechnet. Für diese Methode ließe sich eine Vorbedingung (`side>=0`) angeben, welche prüft, ob die Kantenlänge positiv ist. Diese Vorbedingung sagt aus, dass der Aufruf der Methode mit negativem Argument einen Fehler darstellt. Um die fälschliche Behandlung negativer Kantenlängen zu vermeiden, sollte das Programm um Code erweitert werden, der prüft, ob die Kantenlänge positiv ist. Wer ist aber für die Überprüfung der Kantenlänge verantwortlich? Der Aufrufer der Methode oder die Methode selbst? Die Vorbedingung sagt explizit, dass der *Aufrufer* für den korrekten Aufruf verantwortlich ist. Ohne diese explizite Aussage über die Zuordnung der Verantwortung kann entweder zu wenig überprüft werden (weil Aufrufer und Methode beide annehmen, der andere wäre verantwortlich) oder zu viel (die Überprüfung wird sowohl vom Aufrufer als auch von der Methode durchgeführt). Wird zu wenig geprüft, kann dies zu Fehlern im Programm führen. Zu viel Prüfung führt zu doppelten Überprüfungscode, was die Komplexität des Programmes signifikant erhöhen kann.
Die explizite Angabe einer Vorbedingung hilft, die Komplexität zu reduzieren. Die Gefahr, der Aufrufer könne vergessen, die Parameter zu überprüfen wird durch Überprüfung der Vorbedingung während des Entwicklungsprozesses minimiert.

2.1.1.2 Nachbedingungen

Die Nachbedingung beschreibt, was nach Ausführung einer Operation erwartet wird. Zum Beispiel wird von der Methode zur Berechnung des Flächeninhalts eines Quadrates erwartet, dass $result = side * side$, wobei *result* den Rückgabewert der Methode und *side* die Seitenlänge beschreibt. Die Nachbedingung definiert, was erwartet wird, nicht aber wie dies zu realisieren ist. Sie kann auch als abstrakte Beschreibung der Funktion einer Methode angesehen werden.

2.1.1.3 Invarianten

Eine Invariante ist eine Bedingung, die für eine gesamte Klasse gilt. Beispielsweise sagt eine Invariante einer Klasse, die ein Bank-Konto repräsentiert, dass $kontostand = summe(eingaenge, ausgaenge)$. Eine Invariante gilt 'immer' und für alle Instanzen der Klasse. 'Immer' bedeutet hier: Jeder Zeitpunkt, in dem die Klasse bereit ist, eine Operation auszuführen. Praktisch heißt dies, dass die Invariante beim Aufruf und Beenden jeder öffentlichen Methode geprüft wird. Sie darf während der Abarbeitung einer Methode kurzzeitig ungültig werden, muss aber wieder erfüllt sein, sobald erneut auf eine öffentliche Methode zugegriffen wird.

2.1.2 Vererbung

Das Konzept des *Programmierens mit Vertrag* definiert das Verhalten der Bedingungen bei Vererbung. Es sei zum Beispiel eine Klasse `A` gegeben, welche eine Methode `put()` implementiert. Die Klasse `B` erbt von dieser Klasse und überschreibt die Methode `put()`. Die Neuimplementierung von `put()` birgt potentielle Fehlerquellen, da die Semantik der Methode in der abgeleiteten Klasse verändert sein könnte. Beim Konzept des *Programmierens mit Vertrag* werden die Bedingungen aus der Superklasse (`A`) an alle abgeleiteten Klassen (`B`) vererbt. Vorbedingungen werden in der abgeleiteten Klasse abgeschwächt und Nachbedingungen verstärkt.

2.1.3 Erweiterung durch Trace-Zusicherungen

Eine Trace bezeichnet einen Ablauf eines Programmes und wird meist durch Methodenaufruf-Ketten dargestellt. Beispielsweise sollte in einem Java-Applet zur Initialiasierung zunächst die `init()`-Methode aufgerufen werden und erst anschließend die `start()`-Methode, um mit der Ausführung zu beginnen. Die klassischen Konstrukte des *Programmierens mit Vertrag* (Vor- und Nachbedingung, Invariante), wie sie von Meyer (1998) beschrieben werden, lassen keine Validierung von Traces zu. In den Arbeiten von Fischer (2000) und Plath (2000) wird das Konzept des *Programmierens mit Vertrag* um die Prüfung von Traces erweitert. Eine prototypische Implementierung wurde von Plath (2000) als Erweiterung das Tools Jass (Java with Assertions) im Rahmen seiner Diplomarbeit durchgeführt (siehe auch Abschnitt 2.2.2.4).

2.2 Erweiterungen der Programmiersprache Java

Die Programmiersprache Java verfügt bisher über keine direkte Möglichkeit, das Konzept des *Programmierens mit Vertrag* umzusetzen. In den letzten Jahren wurden einige Ansätze entwickelt, die Java um dieses Konzept erweitern. Diese lassen sich anhand der Techniken, mit denen das Konzept des *Programmierens mit Vertrag* integriert wird, grob untergliedern:

- direkte Erweiterung der Programmiersprache um weitere Schlüsselworte
- Pre-Compiler, der Java Quelltext um Code zur Prüfung von Zusicherungen erweitert
- Bytecode-Instrumentierung zur Erweiterung des Bytecodes um Zusicherungen

- Debugging-Werkzeuge

Dieses Kapitel gibt einen Überblick über aktuelle Ansätze und Tools. Dabei werden folgende Punkte beleuchtet:

1. Entwickler des Tools (kommerziell / frei verfügbar / Prototyp / etc.)
2. Wird das Produkt werterentwickelt?
3. Wie wird Java erweitert?
4. Welche Zusicherungen können modelliert werden?
5. Wird Vererbung unterstützt?
6. Lassen sich Zusicherungen für Interfaces definieren?
7. Können Traces spezifiziert werden?
8. Einbettung in Entwicklungsumgebungen (IDE)?
9. Besonderheiten?

Bemerkung: Die Dokumentation der Tools ließ nicht immer die Beantwortung aller Fragen zu.

2.2.1 Direkte Erweiterung der Programmiersprache

Die Programmiersprache Java kann durch Elemente des *Programmierens mit Vertrag* erweitert werden, indem neue Schlüsselwörter eingeführt werden.

2.2.1.1 JDK 1.4

Bisher wird das Konzept des *Programmierens mit Vertrag* nicht direkt von der Programmiersprache Java unterstützt. In der Version 1.4 des *JDK* (”Java Developer Kit”) wird das neue Schlüsselwort *assert* eingeführt, welches *Assertions* (eine Abwandlung von Exceptions) definiert. Eine Assertion hat folgende Syntax:

```
assert expr1:expr2;
```

Dabei ist `expr1` ein boolscher Ausdruck, der die eigentliche Zusicherung darstellt. `expr2` ist ein optionales Argument, dass zur genaueren Beschreibung des Fehlers herangezogen werden kann.
Durch Platzierung des *assert*-Ausdrucks am Anfang und am Ende der Methode

können Vor- und Nachbedingungen ausgedrückt werden. Diese Bedingungen werden nicht an abgeleitete Klassen vererbt, falls diese die Methode überschreiben. Zusicherungen in Interfaces (sie enthalten keinen Code) sind ebenfalls nicht möglich. Weiterhin ist zu beachten, dass Java-Bytecode mit *Assertions*, der mit dem Compiler des JDK 1.4 erzeugt wurde, nicht mehr in älteren Java Virtual Machines lauffähig ist.(Zörner 2001) (Pöschmann 2001b)

2.2.1.2 AspectJ

AspectJ ist ein Projekt zur *aspektorientierten Programmierung* am *Xerox Palo Alto Research Center*. In der aspektorientierten Programmierung wird versucht, Verhaltenseigenschaften (Persistenz, Logging, Speicherverwaltung, etc.) eines Programmes, die häufig nicht an funktionale Komponenten gebunden sind und über viele Komponenten hinweg verteilt sein können, zu kapseln (siehe AOSD steering committee (2001)). Diese Verhaltenseigenschaften werden *Aspekte* genannt.
AspectJ erlaubt, durch so genannte *Join Points* aufgrund spezieller Ereignisse des ausgeführten Programmes (zum Beispiel Start/Ende einer Methode, Lese- oder Schreibzugriff auf Variablen), zusätzlichen Programmcode auszuführen. Auf Grundlage dieser *Join Points* ließe sich beispielsweise ein Persistenzmechanismus entwerfen, der nach jedem schreibenden Zugriff auf eine bestimmte Variable deren Wert in eine Datenbank schreibt.
Join Points lassen sich auch zur Modellierung von Vor- und Nachbedingungen sowie Invarianten nutzen: Für relevante Methoden und Klassen werden *Join Points* definiert, an welchen Code ausgeführt wird, der die Zusicherungen überprüft. Vererbung von Zusicherungen wird durch geeignete Wahl der *Join Points* unter Verwendung des Sprachkonstruktes *instanceof()* realisiert. Auf diese Weise lassen sich Zusicherungen für bestimmte Klassen und alle davon erbenden Klassen definieren.
Zur Übersetzung von Java-Quelltext, der durch AspectJ Schlüsselwörter erweitert ist, ist ein spezieller AspectJ-Compiler erforderlich. Die aktuelle Version von AspectJ umfasst neben dem Compiler einen Debugger, eine JavaDoc-Erweiterung, sowie Plug-Ins zur Integration von AspectJ in Entwicklungsumgebungen wie Borland JBuilder, Sun Forte oder Emacs. (Kiczales, Hilsdale, Hugunin, Kersten, Palm, und Griswold 2001) (AspectJ Homepage 2001)

2.2.2 Pre-Compiler

Zur Erweiterung von Java kann ein Pre-Compiler eingesetzt werden, welcher die Zusicherungen, die meist in JavaDoc-Kommentaren des Quelltextes untergebracht sind, in Java-Quelltext übersetzt. Diese Erweiterung des Quelltextes wird auch als

Quellcode-Instrumentierung bezeichnet.
Die Erweiterung von Java durch Pre-Compiler hat folgende Vorteile:

1. Die Abbildung der Zusicherungen auf Java-Quellcode kann nachvollzogen werden. Dies erleichtert Software-Entwicklern, Zusicherungen zu spezifizieren. Dieser kann die Übersetzung der Spezifikation im instrumentierten Java-Quelltext einsehen und kontrollieren.

2. Der instrumentierte Quellcode kann mit jedem Java-Compiler in Bytecode überführt werden.

3. Werden die Zusicherungen innerhalb der JavaDoc-Kommentare spezifiziert, kann der Quelltext auch ohne Zusicherungen (und somit ohne Anwendung eines Pre-Compilers) direkt mit einem Java-Compiler übersetzt werden.

4. Bei geeigneter JavaDoc-Konfiguration (angepasstes Doclet) können die Zusicherungen automatisch zur Dokumentation der Implementierung dienen.

Zu den Nachteilen zählen:

1. Das Programm wird durch Einfügen der Zusicherungen verändert. Durch Seiteneffekte in den Zusicherungen kann sich der Ablauf des instrumentierten Programmes von dem des nicht instrumentierten unterscheiden.

2. Häufig werden externe Programmbibliotheken, für welche kein Quellcode zur Verfügung steht, eingesetzt. Bei fehlendem Quellcode kann mit einem Pre-Compiler keine Erweiterung um Zusicherungen durchgeführt werden.

2.2.2.1 iContract

Das freie Tool *iContract* wird von *Reto Kramer* entwickelt und bietet Vor- und Nachbedingungen sowie Invarianten, die sowohl in Java-Interfaces definiert werden können, als auch vererbt werden. Darüber hinaus existieren *forall*- und *exists*-Anweisungen, welche eine Klausel definieren, die auf alle Elemente einer Sammlung anwendbar sein muss. Die Zusicherungen werden in JavaDoc-Kommentaren hinterlegt und durch den iContract Pre-Compiler in Java-Quelltext überführt. Die Integration von iContract in *IBM Visual Age for Java* wird unterstützt. Weiterhin wird iContract durch folgende Tools erweitert:

iControl Grafische Benutzungsoberfläche, welche die Auswahl der zu instumentierenden Klassen erleichtert.

iDoclet JavaDoc Formatvorlage, die neben Standard-Attributen auch die Zusicherungen in der JavaDoc-Dokumentation ausgibt.

iDarvin Ein in Visual Age for Java integrierbares Tool, dass Softwareentwickler bei der Pflege und Erweiterung von Programmen unterstützt.

Weitere Informationen sind in (Pöschmann 2001a), (Kramer 2001) und (Enseling 2001) zu finden.

2.2.2.2 Jcontract

Jcontract ist Bestandteil des kommerziellen Test-Tools *Jtest* der kalifornischen Firma *ParaSoft*. Die Elemente für das *Programmieren mit Vertrag* werden als Kommentar vor der Methode platziert. Der Pre-Compiler wandelt die Zusicherungen aus den Kommentaren in Java-Quelltext um. Vererbung und Spezifikation von Zusicherungen für Interfaces werden unterstützt, Traces jedoch nicht. Wird der instrumentierte Code ausgeführt, können eventuelle Verletzungen der Vertrages entweder in einem Fenster der mitgelieferten grafischen Oberfläche anzeigt oder in eine Textdatei geschrieben werden. Eine IDE-Integration in populäre Entwicklungsumgebungen ist derzeit nicht verfügbar. (Pöschmann 2001a)

2.2.2.3 JML

Die *Java Modeling Language* (JML) ist eine Sprache zur Spezifikation des Verhaltens von Java-Klassen. Sie verknüpft die Konzepte von *Eiffel* und *Larch* mit einigen Elementen des *Refinement Calculus*.
JML wird in Kooperation mit folgenden Projekten entwickelt:

- Das "*Extended Static Checking for Java*"-Projekt am *Compaq Systems Research Center* entwickelt den *Compaq Extended Static Checker for Java* (ESC/Java). Der Static Checker nutzt eine Teilmenge von JML, um den Programmablauf zu spezifizieren und Software-Fehler durch statische Analyse des Codes zu finden.

- An der *Iowa State University* werden in einer Gruppe um *Gary Leavens* zwei Tools entwickelt:

  1. Ein Tool zur automatischen Generierung von Java-Dokumentation unter Berücksichtigung der in JavaDoc-Kommentaren untergebrachten Zusicherungen und deren Vererbung.
  2. Ein Tool zur Prüfung von Vor- und Nachbedingungen sowie Invarianten zur Laufzeit. Zusicherungen werden vererbt und können in Java-Interfaces spezifiziert werden.

- Im *LOOP*-Projekt von *Bart Jakobs* am *Institut für Informatik und Informationstechnik in Nijmegen, Niederlande*, wird die Semantik eines Java-Programmes mit Modelcheckern analysiert. Bisher wird nicht der gesamte Sprachumfang von Java betrachtet, sondern nur eine Teilmenge namens *JavaCard*.

- Das *"Daikon invariant detector"*-Projekt von *Michael D. Ernst* am *Massachusetts Institute of Technology* beschäftigt sich mit der automatischen Generierung von Invarianten aus der Datenstruktur von Java-Programmen.

Die Spezifikation der Traces eines Programmes ist mit der Java Modelling Language nicht möglich. Für weitere Informationen sei auf (Leavens 2001) und (Leavens, Leino, Poll, Ruby, und Jacobs 2000) verwiesen.

2.2.2.4 Jass

Das Tool Jass ist im Rahmen der Diplomarbeiten von Bartetzko (1999) und Plath (2000) an der *Carl von Ossietzky Universität Oldenburg* entstanden und wird weiterhin gewartet (Jass 2001). Die Doktorarbeit von Fischer (2000) behandelt, wie aus einer formalen *CSP-OZ*-Spezifikation Jass-Zusicherungen generiert werden.
Der Jass-Pre-Compiler übersetzt die in JavaDoc-Kommentaren untergebrachten Zusicherungen in Java-Quelltext. Neben den klassischen Elemente des *Programmierens mit Vertrag* (Vor- und Nachbedingungen sowie Invarianten) werden weitere Konstrukte unterstützt:

1. Das Schlüsselwort *check* ermöglicht die Überprüfung von Zusicherungen innerhalb des Methodenrumpfes. Dies ähnelt dem von JDK 1.4 bereitgestellten Assertions (siehe 2.2.1).

2. Mit Hilfe von *rescue* und *retry* kann auf Fehler reagiert oder können Methoden neu gestartet werden.

3. Interferenzen, die bei der Überprüfung von Zusicherungen in Programmen mit mehreren parallelen Threads entstehen können, werden berücksichtigt.

4. Weiterhin können Zusicherungen über das dynamische Verhalten eines Programmes spezifiziert werden. *Trace-Zusicherungen* (englisch: "Trace-Assertions") ermöglichen die Prüfung der Reihenfolge von Methodenaufrufen.

Umsetzung von Trace-Assertions: Zur Erfassung einer Trace fügt Jass (an jedem Methoden-Anfang und jedem Methoden-Ende) Code in den Java-Quelltext

ein, der zur Laufzeit eine zentrale Klasse über den aktuellen Programmablauf informiert. Die erlaubten Sequenzen von Methodenaufrufen werden durch ein Transitionssystem repräsentiert und von Jass in einer Anzahl von Java-Klassen mit dem Präfix `JassTA` abgelegt. Anhand der Informationen über den Programmablauf wird durch das Transitionssystem navigiert und gegebenenfalls ein Fehler angezeigt.

Einschränkungen: Die Implementierung der Trace-Assertions in der aktuellen Version 2 des Jass-Tools unterstützt nicht die volle Funktionalität der Programmiersprache Java:

- Java-Threads werden nicht berücksichtigt.
- Exceptions in den untersuchten Programmen werden nicht unterstützt.

In seltenen Fällen wird bei der Instrumentierung des Quelltextes Java-Code erzeugt, der sich nicht mehr übersetzen lässt und manuelle Nacharbeit erfordert.

Für weitere Informationen zu Jass siehe (Bartetzko, Fischer, Möller, und Wehrheim 2001), (Jass 2001), (Bartetzko 1999), (Plath 2000) und (Fischer 2000).

2.2.3 Bytecode-Instrumentierung

Java kann auch auf Bytecode-Ebene um Elemente des *Programmierens mit Vertrag* erweitert werden. In den originalen Bytecode wird Code für die Überprüfung der Zusicherungen eingefügt. Dieser Vorgang wird als Bytecode-Instrumentierung bezeichnet.
Zu den Vorteilen zählen:

1. Der Java-Quellcode ist nicht erforderlich.
2. Modifikationen des Bytecodes können sowohl vor Ausführung des Programmes auf den Class-Dateien als auch zur Laufzeit durch angepasste Mechanismen zum Laden und Modifizieren der Klassen ausgeführt werden.
3. Werden Änderungen an den Zusicherungen vorgenommen, müssen die Java-Quellen nicht neu übersetzt werden. Es reicht aus, die Modifikationen des Bytecodes anzupassen.

Dieses Konzept birgt jedoch auch folgenden Nachteil:

- Das Programm wird modifiziert. Durch Seiteneffekte der Modifikationen kann sich das Verhalten des modifizierten Programmes vom originalen unterscheiden.

2.2.3.1 jContractor

Mit der Klassenbibliothek *jContractor* der *University of California* werden Vor- und Nachbedingungen sowie Invarianten durch zusätzliche Methoden mit einer speziellen Namensgebung realisiert. Wird eine Klasse mit dem jContractor Class Loader geladen (siehe auch Abschnitt 3.1.4.3) oder mit der jContractor Factory instantiiert, werden die jContractor-Methoden mit Hilfe von Java Reflection lokalisiert und den zu ladenden Bytecode entsprechend instrumentiert. Für jContractor-Methoden gilt folgende Namenskonvention:

Vorbedingung

```
protected boolean
<Name der Methode>_preCondition(<Liste der Argumente>)
```

Nachbedingung

```
protected boolean
<Name der Methode>_postCondition(<Liste der Argumente>)
```

Invariante

```
protected boolean
<Name der Klasse>_classInvariant()
```

Diese Methoden können entweder in die zu erweiternde Klasse eingefügt oder in einer externen Klasse mit dem Namen `<Name der zu erweiternden Klasse>_contract` implementiert werden. Auf diese Weise können Vor- und Nachbedingungen auch für Interfaces definiert werden. Vererbung von Vor- und Nachbedingungen wird unterstützt, Traces jedoch nicht. Weitere Informationen sind in (Karaorman, Hölzle, und Bruno 1998) zu finden.

2.2.3.2 Handshake

Das Tool *Handshake* der *University of California* stellt einen eigenen Mechanismus bereit, um den zu ladenden Bytecode zu instrumentieren. Anders als bei jContractor wird kein eigener Class Loader implementiert, sondern eine native Bibliothek bereitgestellt, welche die ”Datei öffnen”-Aufrufe der Java Virtual Machine (JVM) an das Betriebssystem abfängt. Sobald die JVM versucht, eine Klasse zu laden, ruft sie statt der Betriebssystem-Funktion die ”Datei öffnen”-Prozedur der Handshake Bibliothek auf. Diese reicht den Aufruf an das Betriebssystem weiter und instrumentiert den zurückgelieferten Bytecode-Datenstrom. Im Gegensatz zur Instrumentierung in einem benutzerdefinierten Class Loader können sowohl System-Klassen aus dem `java.lang`-Paket (diese werden immer mit dem Bootstrap Class Loader geladen, siehe 3.1.4.3) als auch

Programme, die einen eigenen Class Loader verwenden, modifiziert werden.

Der Vertrag wird in Text-Dateien spezifiziert, welche mit einem speziellen Compiler in Java-Bytecode übersetzt wird. Mit Hilfe dieses Bytecodes werden die mit Handshake geladenen Klassen instrumentiert.
Handshake unterstützt Vererbung von Vor- und Nachbedingungen sowie Invarianten als auch deren Spezifikation in Java-Interfaces. Der Fokus bei der Entwicklung von Handshake lag in der Entwicklung einer neuen Technik, um das Konzept des *Programmierens mit Vertrag* in Java zu integrieren. Auf Erweiterungen wie Integration in eine IDE oder Trace-Zusicherungen wurde verzichtet. Weitere Details werden von Duncan und Hölzle (1998) beschrieben.

2.2.3.3 Java-MaC

Java-MaC ist eine prototypische Implementierung des Konzepts *Beobachten und Prüfen* (englisch: Monitoring and Checking) von Kim, Kannan, Lee, Sokolsky, und Viswanathan (2001). Ein zu prüfendes Java-Programm wird mit Java-MaC derart instrumentiert, dass an interessanten Stellen des Programmes (Beginn oder Ende einer Methode oder Änderung einer Variablen) Ereignisse erzeugt werden. Die Auswahl der Ereignisse wird mit der *Primitive Event Definition Language* (PEDL) spezifiziert. Die *Meta Event Definition Language* (MEDL) beschreibt aus einer etwas abstrakteren Sicht Bedingungen für erlaubte oder nicht erlaubte Verhaltensweisen eines Programmes. Dieser Ansatz wird in (Kim, Kannan, Lee, Sokolsky, und Viswanathan 2001) und (Kim 2001) näher erläutert.

2.2.4 Erweiterung von Debuggern

Die Java Virtual Machine bietet eine Debug-Schnittstelle, mit deren Hilfe ein Programmablauf beobachtet und manipuliert werden kann. Java-Programme können durch Elemente des *Programmierens mit Vertrag* erweitert werden, indem die Funktionalität dieser Schnittstelle genutzt wird, was folgende Vorteile hat:

1. Der Code des untersuchten Programmes wird nicht modifiziert.

2. Die Kontrolle der Zusicherungen kann zur Laufzeit ein- und ausgeschaltet werden.

3. Das untersuchte Programm und das überprüfende Programm können auf unterschiedlichen Rechner ausgeführt werden. Dies erleichtert zum Beispiel das Prüfen von Programmen in Systemen mit eingeschränkten Ein- und Ausgabefähigkeiten (zum Beispiel eingebettete Systeme).

Wie bei alle anderen Ansätzen verringert sich aufgrund des zusätzlichen Aufwandes für die Überprüfung der Zusicherungen die Ausführungsgeschwindigkeit des untersuchten Programmes.

2.2.4.1 JMSAssert

Die Zusicherungen werden bei *JMSAssert* von *Man Machine Systems* in JavaDoc-Kommentaren durch die Schlüsselwörter `@pre`, `@post` und `@inv` eingeleitet. JMSAssert extrahiert die Zusicherungen aus dem Quellcode und erstellt Vertragsdateien in JMSScript (eine auf Java basierende Skriptsprache von Man Machine Systems).
Zur Überprüfung von Zusicherungen zur Laufzeit wird der Java-Interpreter durch eine auf das Debug-Interface aufsetzende native Bibliothek erweitert. Diese Bibliothek enthält einen JMSScript-Interpreter, welcher beim Aufruf von Methoden des untersuchten Programmes Code aus den Vertragsdateien ausführt.
JMSAssert unterstützt die Spezifikation von Zusicherungen in Java-Interfaces sowie das Vererben von Zusicherungen. Trace-Zusicherungen können jedoch nicht geprüft werden. Zur näheren Informationen wird auf (Man Machine Systems 2000) verwiesen.

2.2.4.2 Lucent Technologies

Im Rahmen des Forschungsprojektes *RTEEM* von *Lucent Technologies* wurde die Prüfung von Zusicherungen mit Hilfe des Java Debug Interface (JDI) (siehe 3.1.3.3) realisiert. Die *Object Constraint Language* (OCL), eine standardisierte Sprache zur Beschreibung von Objekteigenschaften, wird genutzt, um die Zusicherungen zu spezifizieren. Die Architektur des Prototyps besteht aus fünf Komponenten:

1. Eine *Debugger-Applikation* beobachtet und prüft das untersuchte Programm.

2. Ein *Modul*, welches die in OCL notierte Spezifikation parst und evaluiert. Der OCL-Parser wird mit dem Parser-Generator JavaCC erstellt.

3. Eine *Eingabe-Datei* enthält die Spezifikation.

4. Fehlermeldungen werden von einer *Ausgabe-Datei* aufgenommen.

5. Das zu *prüfende Programm* stellt ebenfalls eine Komponente dar.

Die Debugger-Applikation liest die Spezifikation aus der Eingabe-Datei und baut eine entsprechende Datenstruktur auf. Anschließend ruft sie das zu untersuchende

Programm in einer separaten Virtual Machine auf und konfiguriert diese derart, dass Ereignisse beim Aufruf und beim Beenden der spezifizierten Methoden erzeugt werden. Bei jedem Ereignis werden die Invarianten sowie Vor- und Nachbedingungen der Methode aus der Datenstruktur ausgelesen und mit Hilfe des OCL-Parsers/Evaluators validiert. Die Ergebnisse werden in die Ausgabedatei geschrieben.
Dieses Konzept eignet sich, um die klassischen Elemente des *Programmierens mit Vertrag* (Invarianten sowie Vor- und Nachbedingungen) zu prüfen, Traces werden jedoch nicht behandelt. (Murray und Parson 2000)

Bemerkung: Der Code der Debugger Applikation wird zur Zeit von der Firma *Agere Systems* (früher: *Lucent Technologies*) gewartet aber nicht aktiv weiterentwickelt.

2.2.4.3 Query Debugger

Ein weiterer Ansatz zur Erweiterung von Debuggern wurde in (Lencevicius, Hölzle, und Singh 1997) sowie (Lencevicius 2000) vorgestellt. Hier wird ein Mechanismus entwickelt, der es erlaubt, *Objektbeziehungen* zur Laufzeit eines Programmes zu prüfen. Ähnlich der Anfrage-Sprache SQL für relationale Datenbanken, können Anfragen bezüglich der Referenzierungen von Objekten innerhalb der Virtual Machine gestellt werden. Beispielweise ließe sich somit kontrollieren, ob ein Objekt versehentlich mehrfach referenziert wird.
Andere Ansätze schreiben zur Laufzeit eine *Trace* des Programmes in eine Datenbank. Der Inhalt der Datenbank kann zur Laufzeit oder nach Terminierung des Programmes ausgewertet werden.

Weitere Informationen zu erweiterten Debugging-Techniken sind in (AADEBUG 2000) zu finden.

2.2.5 Zusammenfassung

Das von Meyer 1998 eingeführte Konzept des *Programmierens mit Vertrag* ist für die Programmiersprache Java bereits mehrfach umgesetzt (siehe oben). Die einzelnen Ansätze lassen sich anhand der Art und Weise charakterisieren, mit der die Programmiersprache Java erweitert wird: Direkte Spracherweiterung, Pre-Compiler, Bytecode-Instrumentierung und Debugging-Werkzeuge. Die grundlegenden Eigenschaften der einzelnen Ansätze werden nachfolgend zusammengefasst:

- Bei der Erweiterung von Java unter Verwendung eines Pre-Compilers ist der Vertrag meist in JavaDoc Kommentaren der Methoden untergebracht. Ein Softwareentwickler sieht sofort die zur Methode und zur Klasse gehörigen Zusicherungen. Durch angepasste Verarbeitung der JavaDoc-Kommentare können Zusicherungen automatisch in die Dokumentation aufgenommen werden.

- Viele der obigen Tools definieren die Zusicherungen innerhalb der Quelltext Dateien. Für Vor- und Nachbedingungen sowie Invarianten scheint dies ein geeigneter Platz zu sein. Traces beschreiben meist Abläufe, die mehrere Klassen betreffen. Hier ist eine separate Spezifikationsdatei sinnvoller.

- Die Erweiterung mit Hilfe eines Pre-Compilers oder der direkten Spracherweiterung setzt voraus, dass der Java-Quelltext verfügbar ist.

- Durch die direkte Spracherweiterung, den Einsatz von Pre-Compilern oder die Instrumentierung des Bytecodes wird das untersuchte Programm modifiziert. Aufgrund eventueller Seiteneffekte des modifizierten Codes kann sich das modifizierte Programm anders verhalten als das nicht modifizierte.

- Änderungen des Vertrages oder das Ein- beziehungsweise Ausschalten der Vertragsprüfung zieht meist eine Neuübersetzung des untersuchten Programmes nach sich. Debugging-Werkzeuge kommen ohne Neuübersetzung des Programmes aus.

- Das Einfügen zusätzlichen Codes für die Überprüfung der Zusicherungen erhöht die Komplexität des Programmes. Unter Umständen können sich so weitere Fehler einschleichen (z.B.: Erzeugter Quelltext lässt sich aufgrund nicht erreichbarer Codeblöcke nicht mehr übersetzen oder das Programm übersteigt die zulässige Größe und kann von der Virtual Machine nicht mehr ausgeführt werden).

- Allen Ansätze haben einen Nachteil, der sich besonders in zeitkritischen Programmfragmenten bemerkbar macht: Die Ausführungsgeschwindigkeit verringert sich durch die Prüfung der Zusicherugen.

Fazit: Die perfekte Java Erweiterung gibt es nicht. Alle Ansätze und Tools haben ihre Vor- und Nachteile. Die Wahl eines Debuggers zur Überprüfung der Zusicherungen erscheint am besten geeignet:

- Der Quelltext ist nicht notwendig. Klassenbibliotheken fremder Hersteller, deren Quellcode nicht zugänglich ist, können nachträglich um Zusicherungen erweitert werden.

- Der Code des zu prüfenden Programmes unterscheidet sich nicht von dem ausgelieferten.

- Veränderungen des Vertrages oder das Ein- beziehungsweise Ausschalten der Vertragsprüfung ziehen keine Neuübersetzung oder Neuinstrumentierung nach sich. Der Einsatz eines Java Debuggers erlaubt die Prüfung von Programmen auf anderen Rechnern. Zur Laufzeit kann die Kontrolle bestimmter Zusicherungen individuell konfiguriert werden.

Die Studie von Lucent Technologies (siehe Abschnitt 2.2.4.2) belegt die Machbarkeit der Kontrolle von Zusicherungen unter Einsatz eines speziellen Debuggers. Frei verfügbare Programme, die das Konzept des *Programmierens mit Vertrag* für Java mit Hilfe eines angepassten Debuggers umsetzen, sind zur Zeit nicht vorhanden.

2.3 Exkurs: CSP und Java

In diesem Kapitel wurden bisher Werkzeuge vorgestellt, die die Programmiersprache Java um das Konzept *Programmieren mit Vertrag* erweitern. Alle Werkzeuge prüfen zur Laufzeit eines Programmes, ob ein Vertrag eingehalten wird oder nicht. Ein anderer Ansatz zur Erhöhung der Korrektheit ist der Einsatz von Klassenbibliotheken wie CTJ, JCSP oder Jack. Diese erweitern die Programmiersprache Java um Elemente der theoretisch weit erforschten Sprache CSP (”Communicating Sequential Prozesses”). Die Erfahrungen und Softwarewerkzeuge aus zwanzig Jahren Forschung über CSP werden somit auch in Java nutzbar. (weitere Informationen zu CSP sind in Kapitel 3.2 zu finden). Die Bibliotheken erleichtern unter anderen das Erstellen von Applikation mit mehreren parallelen Threads. Der Programmierer muss sich nicht mehr direkt mit der Realisierung und Synchronisation von Java Threads auseinandersetzen, sondern kann auf eine abstraktere CSP-ähnliche Struktur aufsetzen. Die einzelnen Bibliotheken werden nun näher beschrieben:

CTJ

CTJ (”Concurrent Threads in Java”) wird an der *University of Twente, Niederlande* im Rahmen des *JavaPP*-Projektes entwickelt. Die Bibliothek implementiert ein CSP-basiertes Thread-Model für Java, wobei ein eigener Mechanismus zur Realisierung von Parallelität und Synchronisation eingesetzt wird. *CT* ist hauptsächlich für den Bereich von Realzeit-Anwendungen konzipiert. (Bakkers 2001)

JCSP

Die Bibliothek *JCSP* (”Communicating Sequential Processes for Java”) wird von *Peter Welch* und *Paul Austin* an der *University of Kent, Canterbury* entwickelt. Sie bietet einen Alternative zum Monitor-Konzept zur Verwaltung von Threads und Synchronisation in Java. Parallelität und Synchronisation werden mit Hilfe von abstrakteren CSP-Mechanismen modelliert, welche jedoch intern auf dem Monitor-Konzept von Java aufbauen.

Für die einwandfreie Funktionalität der *JCSP*-Bibliothek sollten die Mechanismen von *JCSP* und Java nicht gemischt werden: Beim Einsatz von *JCSP* sollte auf synchronisierte Methoden, Instanzen von `java.lang.Runnable` und `java.lang.Thread` oder Aufrufe der `wait()`, `notify()` oder `notifyAll` Methoden der Klasse `java.lang.Object` verzichtet werden. Weitere Informationen werden von Welch (2001) beschieben.

Jack

Im Rahmen der Diplomarbeit von Freitas (2002) an der *Federal University of Pernambuco - UFPE, Brazil* wird das Framework *Jack* entwickelt, welches Java um CSP-Elemente erweitert. Dieses unterscheidet sich von *CTJ* und *CSPJ* hauptsächlich durch seine Erweiterbarkeit und einer konsequenteren Trennung einzelner Aufgaben. Die Arbeit von *Leonardo Freitas* wird voraussichtlich im Frühjahr 2002 veröffentlicht.

2.4 Jassda

In diesem Kapitel wurden einige Ansätze und Tools vorgestellt, die die Programmiersprache Java um das Konzept des *Programmierens mit Vertrag* erweitern. Sie erlauben die Prüfung von Zusicherungen zur Laufzeit. Trace-Assertions werden bisher wenig berücksichtigt. Aussagen über das zeitliche Verhalten von Programmen werden von den vorgestellten Ansätzen und Tools nicht unterstützt.
Im Rahmen dieser Diplomarbeit wird eine erweiterbare Architektur entworfen, welche die Prüfung von Trace- und Zeit-Zusicherungen von Java-Programmen zur Laufzeit erlaubt. Diese Architektur wird **J**ava with **Ass**ertions **D**ebugger **A**rchitecture (oder kurz: Jassda) genannt. Um Trace- und Zeit-Zusicherungen zur Laufzeit prüfen zu können, sind folgende Aufgaben zu lösen:

1. Die Trace eines Programmes ist zu ermitteln.
2. Die ermittelte Trace muss geprüft werden.

Nachfolgend wird beschrieben, wie Jassda die einzelnen Aufgaben löst:

2.4.1 Ermittlung der Trace

Die Trace (oder Spur) eines Programmes wird durch eine Sequenz von Methodenaufrufen charakterisiert. Es gilt also, die Abfolge von Methodenaufrufen zu erfassen. Dazu können Techniken genutzt werden, die bereits von den oben vorgestellten Tools eingesetzt werden: Statt Vor- und Nachbedingungen oder Invarianten am Anfang und am Ende der Methoden zu prüfen, wird an diesen Stellen über den Programmablauf informiert.

Wie in der Zusammenfassung (siehe Abschnitt 2.2.5) beschrieben, ist die Erweiterung von Java um das Konzept des *Programmierens mit Vertrag* unter Verwendung spezieller Debugger am flexibelsten: Der Zugriff auf den Quelltext ist nicht notwendig. Es wird genau der Code untersucht, welcher auch ausgeliefert wird. Zusicherungen sind nicht im Code des Programmes integriert und lassen sich komfortabel zur Laufzeit ein- und ausschalten.
Um die Trace eines Programmes zu ermitteln setzt Jassda auf das Java Debug Interface (JDI) auf, welches in Abschnitt 3.1.3.3 näher beschrieben wird.

2.4.2 Prüfen einer ermittelten Trace

Die Sprache CSP (”Communicating Sequential Processes”) eignet sich zur Beschreibung, Validierung und Verifikation von Traces. Sie wurde vor mehr als 20 Jahren von Hoare (1978) entwickelt und verfügt über eine genau definierte Semantik. Mit formalen Methoden lassen sich CSP-Spezifikationen auf Deadlocks etc. untersuchen. Der Exkurs 2.3 zeigt, dass CSP bereits Einzug in die Java-Welt gehalten hat. Die vorgestellten Bibliotheken helfen bei der *Konstruktion* von Java-Anwendungen.

Zur Spezifikation erlaubter Traces wird auf die weitreichend erforschte Sprache CSP zurückgegriffen. Ein speziell an die Spezifikation von Traces von Java-Programmen angepasster CSP-Dialekt wird in Kapitel 3.2 formal eingeführt.

2.4.3 Architektur

Bei der Entwicklung der *Java with Assertions Debugger Architecture* (siehe Kapitel 4) wurde stark auf eine komponentenbasierte Struktur und hohe Wiederverwendbarkeit geachtet. Die Implementierung von Jassda stellt ein Framework zur Verfügung, welches zur Laufzeit die Trace eines untersuchten Programmes ermittelt. Die Informationen über die Trace werden laufend an ein oder mehrere Module geleitet. Was ein Modul mit der Information anfängt, bleibt ihm überlassen.

2.4.3.1 Framework

Das Framework ermöglicht, eine oder mehrere Programme zu beobachten und Informationen über deren Trace an die Jassda-Module weiterzugeben. Die gelieferten Informationen sind sehr detailliert: Unter anderen kann ermittelt werden, welche Methode welcher Instanz welcher Klasse innerhalb welchem Java Thread aufgerufen wurde. Darüber hinaus implementiert das Jassda-Framework Uhren, die zur Validierung des zeitlichen Verhaltens genutzt werden können.

2.4.3.2 Module

Die folgenden Module nutzen das Framework:

- Ein Logger-Modul schreibt den Ablauf eines Programmes in eine Datei.
- Das Trace-Checker-Modul prüft zur Laufzeit eines Programmes, ob dieses Trace- und Zeit-Zusicherungen einhält. Die Zusicherungen werden in CSP_{jassda} spezifiziert, einem CSP-Dialekt, der in 3.2 formal beschrieben wird.

Bemerkung: Der Begriff Jassda bezeichnet im weiteren Verlauf dieser Arbeit die Implementierung der *Java with Assertions Debugger Architecture*, also das Tool, welches auf das Jassda-Framework aufbaut und die Module *Trace-Checker* und *Logger* nutzt.

2.4.3.3 Funktionsweise

Das Tool Jassda beobachtet den Ablauf eines Programmes, indem es *Breakpoints* an den Anfang und an das Ende von Methoden setzt. Sobald zur Laufzeit ein Breakpoint erreicht wird, unterbricht Jassda die Ausführung des Programmes und erzeugt ein Ereignis. Dieses wird an Module weitergeleitet, welche zum Beispiel den aktuellen Zustand des überprüften Programmes in eine Datei schreiben oder den Ablauf des Programmes gegen eine Spezifikation prüfen. Haben alle Module das Ereignis verarbeitet, wird die Unterbrechung des untersuchten Programmes aufgehoben.

Bemerkung: Jassda ist keine weitere Implementierung des Konzeptes *Programmieren mit Vertrag* für die Programmiersprache Java. Stattdessen bietet Jassda ein *Framework*, mit dessen Hilfe schnell Werkzeuge zur Runtime-Validierung von Java-Programmen erstellt werden können. Weiterhin umfasst

Jassda die oben erwähnten Module zur Protokollierung von Validierung von Traces. Ein Modul, welches Vor- und Nachbedingungen sowie Invarianten prüft, ist denkbar.

Kapitel 3

Grundlagen

Aus der Beobachtung der menschlichen Vorgehensweise bei der Suche nach Software-Fehlern lassen sich hilfreiche Konzepte für die automatische Prüfung von Programmen und speziell der Validierung von Trace- und Zeitzusicherungen zur Laufzeit ableiten. Zwei grundlegende Aufgaben sind zu erfüllen:

1. Die Trace des untersuchten Programmes muss ermittelt werden.

2. Die ermittelte Trace ist zu überprüfen. Wurde diese Trace erwartet?

Im Abschnitt 3.1 wird zunächst beschrieben, welche Techniken einem Software-Entwickler bei der Lokalisierung eines Fehlers helfen können. Anschliessend wird die Java Platform Debugger Architecture (JPDA) vorgestellt, die eine standardisierte Schnittstelle für das Debuggen von Java-Programmen bereitstellt. Um zu verstehen, welche Informationen mit Hilfe dieser Schnittstelle geliefert werden können, wird ein kleiner Einblick in die Architektur der Java Virtual Machine gegeben.

Bei der Suche nach einem Fehler wird der Ablauf eines Programmes mit einem erwarteten Ablauf verglichen. Der Trace-Prüfer, welcher Bestandteil von Jassda ist, vergleicht den konkreten Ablauf eines Programmes mit dem in einer Spezifikation definierten erwarteten Ablauf. Die Spezifikation wird in einem Dialekt der Sprache CSP (”**C**ommunicating **S**equential **P**rocesses”, siehe Bowen 2001) definiert. Dieser Dialekt wird in Abschnitt 3.2 formal eingeführt.

In einigen Anwendungen ist es notwendig oder ein Qualitätsmerkmal, dass Zeitschranken eingehalten werden. Ein Anwender erwartet beispielsweise, dass das Programm auf eine Anfrage innerhalb einer bestimmten Zeit antwortet. Die Spezifikation von Zeit-Zusicherungen wird in Abschnitt 3.3 beschrieben.

3.1 Debugging

Zu den unbeliebtesten Aufgaben eines Software Entwicklers gehört das Testen und Debuggen von Programmen. Es wird nach generellen Problemen gesucht oder versucht, das System zum Abstürzen zu bringen. Häufig ist ein Fehler bekannt und die Ursache muss lokalisiert werden. In großen und komplexen Systemen fällt es schwer, die Ursache anhand kryptischer Fehlermeldungen zu finden. Hier kommen Debug-Techniken zum Einsatz.

Unter ”Debugging” versteht man die Analyse und Bearbeitung eines Programmes, um Bugs zu finden und zu beseitigen. Zwei Debugging-Techniken werden im Folgenden vorgestellt.

3.1.1 Debugging durch Logging

Eine weit verbreitete Debugging-Technik besteht darin, das Programm durch Code-Fragmente zu erweitern, welche an kritischen Stellen den aktuellen Zustand des Programmes ausgeben. Dies kann in Java beispielsweise durch den `System.out.println()`-Befehl erfolgen.
Diese Technik der Fehlersuche erfordert jedoch nach jedem Einfügen einer Debug-Ausgabe eine Neuübersetzung des Programmes. Weiterhin können die Ausgaben schnell sehr umfangreich werden. Diese Methode wird meist nicht durch Software-Werkzeuge unterstützt.

3.1.2 Debugging mit Werkzeugunterstützung

Für die meisten Programmiersprachen existieren Werkzeuge, die die Fehlersuche erleichtern. Diese Debugger erlauben, das zu untersuchende Programm (auch *Debuggee* genannt) schrittweise auszuführen und bei jedem Schritt die Werte der Variablen und andere Daten einzusehen. Für die Programmiersprache Java existieren viele dieser Werkzeuge. Fast jede Java-Entwicklungsumgebung verfügt über einen Debugger.
Die meisten Java Debugger setzen auf die **J**ava **P**latform **D**ebugger **A**rchitecture (JPDA) auf.

3.1.3 Java Platform Debugger Architecture (JPDA)

Seit der Version 1.2 des Java SDK (”Software Developer Kit”) wird mit der *Java Platform Debugger Architecture* (JPDA) eine einheitliche Architektur für den Zugriff auf die Java Virtual Machine durch Debugger bereitgestellt. Diese besteht aus zwei Schnittstellen (JVMDI und JDI), einem Protokoll (JDWP) sowie

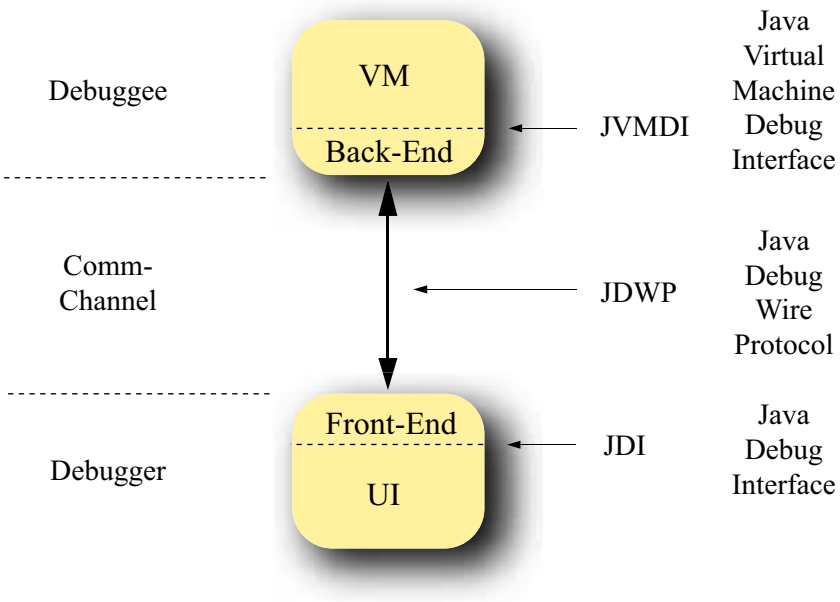

Abbildung 3.1: Java Platform Debug Architecture

zwei Software-Komponenten, die diese Elemente miteinander verbinden(Front-End und Back-End) (siehe Abbildung 3.1). (Sun Microsystems, Inc. 2001)

3.1.3.1 Java Virtual Machine Debug Interface (JVMDI)

Das Java Virtual Machine Debug Interface (JVMDI) beschreibt die Funktionalität einer Virtual Machine (VM), die das Debugging eines in dieser Virtual Machine ablaufenden Java Programmes ermöglicht. In der Referenzimplemtierung der Java Platform Debug Architecture (JPDA) wird das JVMDI als nativer Bestandteil der Virtual Machine implementiert. Auf diesem Interface basiert das Back-End welches die nativen Funktionsaufrufe für ein plattformunabhängiges Protokoll (JDWP) aufbereitet.

3.1.3.2 Java Debugger Wire Protocol (JDWP)

In der JDPA besteht ein Informations-Kanal zwischen dem Front-End (im Debugger-Prozess) und dem Back-End (im Debuggee-Prozess). Das JDWP beschreibt das Format der Daten, die über diesen Kanal ausgetauscht werden.

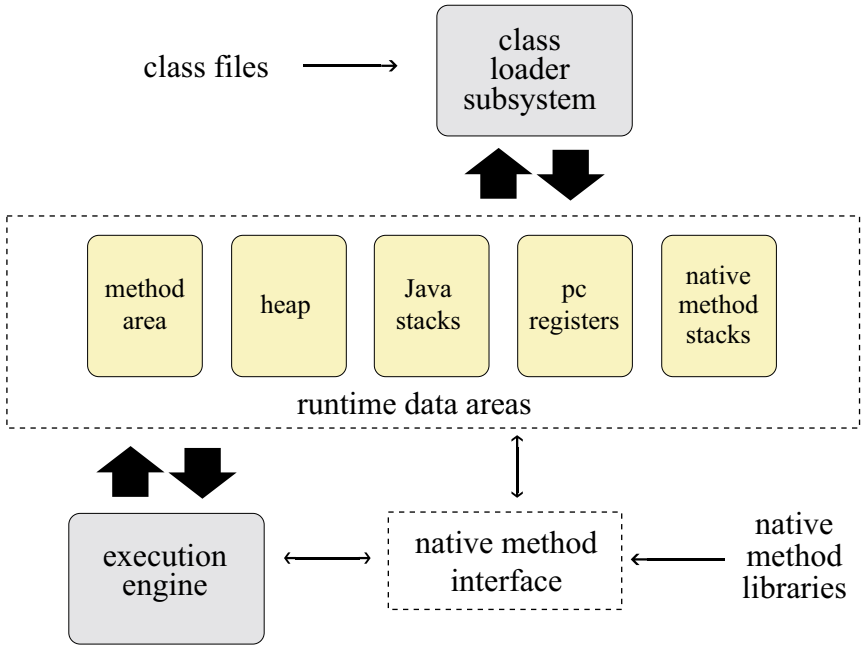

Abbildung 3.2: Architektur der Java Virtual Machine

3.1.3.3 Java Debug Interface (JDI)

Das Java Debug Interface (JDI) bietet eine vollständig in der Programmiersprache Java implementierte Schnittstelle für das Debugging von Java Applikationen. Es wird vom Front-End implementiert und bereitet die Informationen auf, die das JDWP liefert. Diese Java API (”Application Programming Interface”) ermöglicht den komfortablen Zugriff auf einen Debuggee von einer Java Applikation.

3.1.4 Virtual Machine (VM)

Um zu verstehen, welche Informationen vom Java Debug Interface bereitgestellt werden können, ist ein grobes Verständnis über die Funktionsweise der Virtual Machine notwendig. Die Java Virtual Machine Spezifikation (Lindholm und Yellin 1999) beschreibt das Verhalten einer Virtual Machine durch Subsysteme, Speicherbereiche, Datentypen und Befehle. Diese Komponenten beschreiben abstrakt den inneren Aufbau der Virtual Machine. Abbildung 3.2 zeigt die wichtigsten Subsysteme und Speicherbereiche.

class loader Das Classloader Subsystem stellt einen Mechanismus zum Laden von Java-Klassen und Interfaces bereit.

execution engine Die Execution Engine verarbeitet den Bytecode der geladenen Klassen.

runtime data areas In diesem Speicherbereich wird neben dem Bytecode und weiteren Informationen der Class Dateien auch Objekte, die das Programm instantiiert, Parameter von Methoden, Rückgabewerte, lokale Variablen sowie Zwischenergebnisse von Berechnungen abgelegt. Folgende Speicherbereiche sieht die Spezifikation für eine Virtual Machine vor:

Method Area - Die Method Area speichert den Bytecode und Daten über Variablen und Methoden.

Heap - Im Heap werden zur Laufzeit erzeugte Elemente abgelegt.

Java Stacks - Die Java Stacks verwalten (für jeden Thread getrennt) lokale Variablen und Methoden-Parameter.

Pc-Registers Die Programmzähler speichern die aktuelle Position im Bytecode. Jeder Thread hat einen eigenen Programmzähler.

Native Method Stacks - Der native Method Stack stellt Speicherplatz für die Ausführung nativer Methoden bereit. Für jeden Thread, der nativen Code ausführt, existiert ein eigener Native Method Stack.

Einige für Jassda interessante Aspekte der Java Virtual Machine werden nachfolgend genauer erläutert.

3.1.4.1 Method Area

Der als *method area* bezeichnete Speicherbereich wird von allen Threads geteilt und speichert für jede Klasse den Bytecode der Methoden sowie Daten über Variablen und Methoden. Außerdem ist hier der *Runtime Constant Pool* zu finden, welcher den *Constant Pool* der Class-Dateien zur Laufzeit repräsentiert. Für jede Klasse werden hier Information über Konstanten sowie Referenzen zu Methoden und Variablen verwaltet. Nach dem Laden der Klassen beinhaltet der *Runtime Constant Pool* symbolische Referenzen zu Variablen und Methoden. Um die Zugriffsgeschwindigkeit zu erhöhen, werden diese später durch direkte Zeiger auf die Methoden und Variablen ersetzt. (siehe auch 3.1.4.4)

3.1.4.2 Java Stack

Die Virtual Machine verwaltet für jeden Thread ein eigenes Befehlsregister, sowie einen eigenen Java Stack (siehe Abbildung 3.3). Wird innerhalb eines Threads

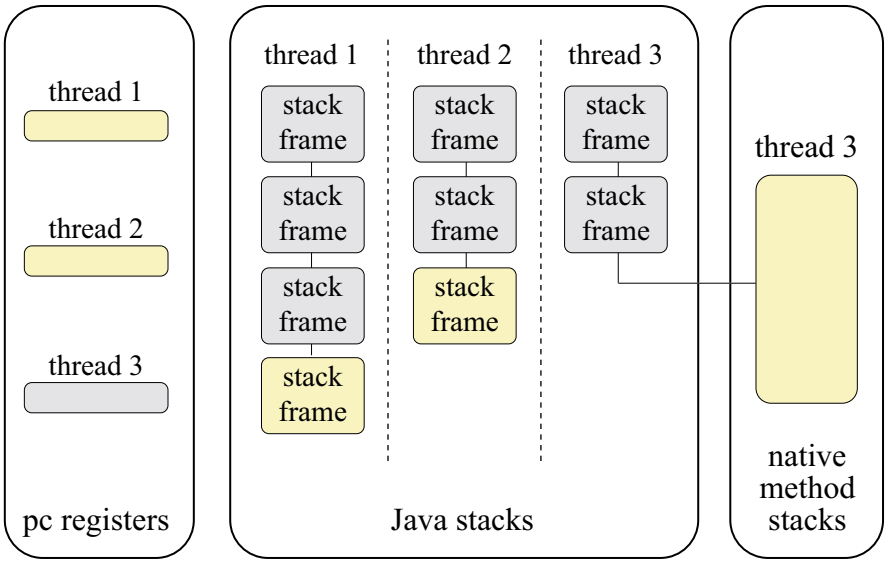

Abbildung 3.3: Java Stacks

eine Java-Methode ausgeführt, enthält das Befehlsregister die Adresse des nächsten Befehls und der Stack speichert die Umgebung (lokale Variablen, Methoden-Parameter, Rückgabewerte, Zwischenergebnisse bei Berechnungen) der Methode. Wird eine native Methode ausgeführt, werden die Daten auf dem Native Method Stack abgelegt.
Der Java Stack ist in einzelne Stack Frames untergliedert. Ein Stack Frame speichert die Umgebung des Aufrufs einer Methode. Sobald eine Methode aufgerufen wird, legt die Virtual Machine im Stack des aktuellen Threads ein neues Stack Frame an. Sobald die Methode beendet ist, wird das dazugehörige Stack Frame wieder vom Stack entfernt. Ein Stack Frame enthält Speicherplatz für lokale Variablen, den *Operand-Stack* sowie weitere Informationen die von der konkreten Implementierung der Virtual Machine benötigt werden. Auf dem *Operand-Stack* werden vor dem Aufruf einer Methode die Methoden-Argumente sowie die Referenz zur Objektinstanz (falls vorhanden) abgelegt. Nach der Ausführung der Methode befindet sich der Rückgabewert auf dem Operand-Stack.

3.1.4.3 Class Loader

Das Class Loader Subsystem dient dem Finden und Laden von Java-Bytecode-Dateien (auch als *Class*-Dateien bezeichnet). Es wird zwischen drei Typen von Class Loadern unterschieden:

Bootstrap Class Loader Der Bootstrap Class Loader ist Teil der Implementie-

rung der Virtual Machine. Der Bootstrap Class Loader von Sun's Java 2 SDK sucht ausschließlich in dem Verzeichnis, in welchem die Java System-Klassen untergebracht sind. Der Wert der Umgebungsvariablen `CLASSPATH` wird nicht ausgewertet. Kann die Klasse mit dem Bootstrap Class Loader nicht gefunden werden, wird als nächstes geprüft, ob die Klasse in einem optionalen Paket (auch als "Standard Extension" oder einfach "Extension" bezeichnet) enthalten ist.

User-defined Class Loader Der benutzerdefinierte Class Loader ist Teil der Applikation. Sun's Java 2 SDK lädt beim Starten der Virtual Machine automatisch den *System Class Loader*. Der System Class Loader berücksichtigt den Classpath der Virtual Machine bei der Suche nach Class-Dateien.

Durch den Mechanismus benutzerdefinierter Class Loader, können Class-Dateien von unterschiedlichen Orten geladen werden (zum Beispiel: FTP, Http, Dateisystem, etc.). Weiterhin können auch die Class-Dateien zum Zeitpunkt des Ladens manipuliert oder gar vollständig neu erstellt werden.
Mehrere benutzerdefinierte Class Loader können parallel eingesetzt werden. Klassen, die von unterschiedlichen Class Loadern geladen wurden, werden intern mit verschiedene Name Spaces versehen.

3.1.4.4 Aufruf von Java-Methoden

Jassda beschreibt die Trace eines Programmes anhand der Abfolge von Methoden-Aufrufen. In diesem Abschnitt wird genauer beschrieben, wie ein Aufruf einer Methode von der Virtual Machine realisiert wird.

Dynamisches Binden: Java Programme werden symbolisch gebunden. Das heißt: die Class-Dateien, die von der Java Virtual Machine geladen werden, enthalten keine direkten Zeiger auf einzelne Methoden, sondern symbolische Referenzen.
Beim ersten Aufruf einer Methode wird anhand des Namens der Klasse und Methode sowie deren Signatur (Anzahl und Typ der Argumente) der Zeiger zum Anfang des Bytecodes der Methode ermittelt und die symbolische Referenz im Runtime Constant Pool durch diesen Zeiger ersetzt.

Verifikation: Bevor der Bytecode der Methode ermittelt wird, wird überprüft, ob die Regeln der Java Sprache eingehalten wurden (z.B.: "existiert der referenzierte Bytecode?") und die Methode aufgrund von Sicherheitsbestimmungen überhaupt aufgerufen werden darf (z.B.: "wird eine private Methode einer anderen Klasse aufgerufen?").

Objektreferenz und Argumente: Wird eine Instanz-Methode (eine nicht als *static* deklarierte Methode) aufgerufen, wird neben der Referenz zum Bytecode auch eine Referenz zu den Daten des Objektes benötigt. Diese Referenz wird vor dem Aufruf der Methode auf dem *Operand-Stack* der aufrufenden Methode abgelegt. Die Argumente der Methode werden ebenfalls auf dem Operand-Stack abgelegt.

Stack Frame: Bei jedem Aufruf einer Methode wird von der Java Virtual Machine ein neues Stack Frame auf dem Stack des aktuellen Threads erzeugt. Die Methoden-Argumente sowie die Objekt-Referenz (falls vorhanden) werden vom Operand-Stack der *aufrufenden* Methode genommen und auf dem Operand-Stack der *aufgerufenen* Methode abgelegt.
Nach Beenden der Methode wird der Rückgabewert auf den Operand-Stack der *aufrufenden* Methode geschrieben und der Stack Frame der beendeten Methode entfernt.

3.1.4.5 Aufruf nativer Methoden

Im Gegensatz zum Aufruf einer Java-Methode wird beim Aufruf nativer Methoden kein neues Stack Frame auf dem Java Stack abgelegt. Stattdessen wird meist ein separater Stack (*native method stack*) verwendet. Die Verarbeitung nativer Methoden ist von der Implementierung der Virtual Machine abhängig.

3.1.4.6 Umgang mit Exceptions

Der Ausdruck *Exception* beschreibt ein ungewöhnliches Ereignis. Das *Java Tutorial* (siehe Campione, Walrath, und Huml 2000a) definiert eine Exception als ein Ereignis, welches bei der Ausführung eines Programmes auftritt und den normalen Programmablauf unterbricht. Ist eine Exception aufgetreten, sucht die Virtual Machine nach Code, welcher auf diese Exception reagiert. Dafür wird in der *Exception-Table* der Methode gesucht, ob für die aktuelle Position im Code entsprechender Code (durch das Schlüsselwort `catch` im Quelltext markiert) zur Behandlung der Exception referenziert wird. Ist dies nicht der Fall, wird die Suche in den nächsten Methoden auf dem Java Stack fortgesetzt.

3.1.4.7 Breakpoints

Breakpoints sind Marken, die Stellen eines Programmes kennzeichnen, an denen dessen Ausführung angehalten werden soll. Die Spezifikation der Java Virtual Machine sieht für die Realisierung von Breakpoints einen spezielle Opcode vor,

der in den Bytecode der Methode eingefügt werden kann. (Lindholm und Yellin 1999)

3.1.5 Java Debug Interface (JDI)

Das Java Debug Interface (JDI) ist eine Java API, die Informationen für Debugger und ähnliche Tools bereitstellt, welche auf die Java Virtual Machine zugreifen. Das JDI ermöglicht den Zugriff auf eine laufende Virtual Machine. Beispielsweise können geladene Klassen und Interfaces sowie erzeugte Instanzen und Arrays untersucht werden. Darüber hinaus kann auch auf den Programmablauf durch Manipulation von Variablen, Aufrufen von Methoden oder durch Anhalten und Starten einzelner oder aller Threads eingewirkt werden. Jassda nutzt das JDI, um Information über den Ablauf eines Programmes zu sammeln.
In den folgenden Abschnitten werden einige Funktionen, die von Jassda genutzt werden, näher erläutert.

3.1.5.1 Verbindung zum Debuggee

Die Verbindung der Debugger-Applikation zum Debuggee kann durch unterschiedliche *Connectors* hergestellt werden. Das Java 1.3 SDK implementiert folgende *Connectors*:

LaunchingConnector Der *LaunchingConnector* startet eine Java Applikation und verbindet sich mit dessen Virtual Machine.

AttachingConnector Ist eine Java Virtual Machine im Debug-Modus gestartet, kann zu ihr eine Verbindung mit dem *AttachingConnector* hergestellt werden.

ListeningConnector Mit dem *ListeningConnector* kann ein Debugger implementiert werden, der auf eine Verbindung wartet, die von der Virtual Machine eines zu prüfenden Programmes initiiert wird.

Attaching- und *ListeningConnector* ermöglichen Remote-Debugging: Debugger und Debuggee können auf verschiedenen Rechnern ausgeführt werden. Ein Debugger kann gleichzeitig mit mehreren Debuggees verbunden sein. Umgekehrt kann in Debuggee jedoch *nicht* gleichzeitig mit mehreren Debuggern verbunden sein.

3.1.5.2 Das Java Interface `VirtualMachine`

Sobald ein *Connector* eine Verbindung zwischen Debuggee und Debugger aufgebaut hat, stellt er ein Objekt bereit, welches das Java Interface `com.sun.jdi.VirtualMachine` implementiert. Dieses Objekt spielt eine zentrale Rolle beim Umgang mit dem Java Debug Interface (JDI). Pro Verbindung zu einem Debuggee existiert ein `VirtualMachine`-Object. Fast jedes Object, dass vom JDI erzeugt wird, enthält eine Referenz auf das `VirtualMachine`-Object.
Das Java Interface `VirtualMachine` ermöglicht unter anderen Zugriff auf:

- eine Liste der geladenen Klassen
- eine Liste aktiver Threads
- den Ausführungszustand - die Virtual Machine kann angehalten und wieder gestartet werden
- einen Mechanismus zum Konfigurieren und Abfragen von Ereignissen (siehe Abschnitt 3.1.5.3)
- Methoden zum Erzeugen primitiver Objekte (Typ boolean, byte, char, short, int, long, float, double, String) in der untersuchten Virtual Machine. Diese Objekte können genutzt werden, um Variablen mit neuen Werten zu belegen.

Weiter Informationen lassen sich indirekt erfragen. `Jassda` benötigt zum Beispiel die Namen der Methoden aller geladenen Klassen. Um die Namen zu ermitteln, wird über `VirtualMachine` eine Liste der geladenen Klassen erfragt. Für jede Klasse kann wiederum eine Liste der Methoden angefordert werden.

3.1.5.3 Ereignisse

Die Java Virtual Machine kann Ereignisse erzeugen, um den Debugger über interne Aktionen zu informieren. Für folgende Aktionen können Ereignisse erzeugt werden:

- Breakpoint ist erreicht
- Variable wird gelesen
- Variable wird modifiziert
- Klasse wird vorbereitet (geladen und verifiziert)
- Klasse wird aus der Virtual Machine entfernt

- Thread gestartet
- Thread beendet
- Exception ist aufgetreten
- Methode wurde aufgerufen
- Methode wurde beendet

3.1.5.4 Einschränkungen des JDI

Das JDI erlaubt weitreichenden Zugriff auf ablaufende Virtual Machines. Eine genauere Betrachtung zeigt jedoch auch Unzulänglichkeiten.

Rückgabewerte von Methoden: Die Implementierung des JPDA in Sun's SDK 1.3 erlaubt keinen direkten Zugriff auf den Rückgabewert einer Methode. Bereits im Dezember 1998 wurde der Wunsch geäußert, das `MethodExitEvent` (wird nach dem Verlassen einer Methode erzeugt) um eine Methode für den Zugriff auf den Rückgabewert zu erweitern (siehe Java Developer Connection, Bug Database 1998). Diese Methode ist bisher nicht implementiert.

Wie in Abschnitt 3.1.4.4 beschrieben, ist der Rückgabewert nach dem Aufruf einer Methode auf dem Operand Stack der aufrufenden Methode abgelegt. Die Java Platform Debug Architecture bietet jedoch keine Möglichkeit, den Operand Stack einzusehen. Eine entsprechende Methode (`GetOperandStack`) ist zwar in der C-Header-Datei `jvmdi.h` des Java Virtual Machine Debug Interfaces aufgeführt, die Implementierung fehlt jedoch. (siehe Quellcode des Sun JDK1.3 in Sun Community Source 2001 sowie Java Developer Connection, Bug Database 2001). In Abschnitt 4.6.1 wird beschrieben, wie durch leichte Modifikation des Bytecodes auf den Rückgabewert zugegriffen werden kann.

Compilierung mit Debug-Option: Um die volle Funktionalität des JDI nutzen zu können, muss das untersuchte Programm Debug-Informationen enthalten. Das Programm muss mit der Debug-Option (Aufruf des Sun Java Compilers mit `javac -g`) kompiliert sein. Andernfalls kann nicht auf Variablen und Methodenparameter zugegriffen werden. Diese Einschränkung betrifft nicht die gesamte Java Platform Debugger Architecture (JPDA), sondern nur das Java Debug Interface (JDI). Sowohl das JVMDI als auch das JDWP ermöglichen den Zugriff auf Variablen, auch wenn der entsprechende Code keine Debug-Informationen enthält.

Kapitel 4.6.2 beschreibt, wie Class-Dateien nachträglich um Debug-Informationen erweitert werden.

3.2 Spezifikation von Traces

Die Trace eines Programmes kann durch Beschreibung der Abfolge seiner Methoden und der jeweils gültigen Werte der Daten (Variablen, Rückgabewerte) spezifiziert werden. Zur Spezifikation von Traces in komplexen Systemen eignet sich die Sprache der *Communicating Sequential Processes* (CSP) (eine Einführung in CSP ist unter anderen in (Roscoe 1997), (Hoare 1985) und (Schneider 2000) zu finden). Im Rahmen dieser Arbeit wird der CSP-Dialekt CSP_{jassda} entwickelt, der speziell für die Beschreibung von Trace- und Zeit-Zusicherungen in Java-Programmen angepasst ist. Dieses Kapitel gibt zunächst einen kurzen geschichtlichen Überblick über CSP und beschreibt dann formal die Sprache CSP_{jassda} .

3.2.1 Einführung

1985 entwickelte Charles Antony Richard Hoare eine Sprache zur Spezifikation von Systemen unabhängiger, mit der Umgebung kommunizierender Komponenten. Die Idee ist, dass das System in Subsysteme (Komponenten) zerlegt werden kann, welche gleichzeitig operieren und miteinander als auch mit der Umgebung interagieren.

Ein *Prozess* symbolisiert das Verhalten einer Komponente. Das Verhalten wird durch abstrakte Ereignisse (englisch: Events) beschrieben, welche die Prozesse leiten. An jedem möglichen Schritt bei der Abarbeitung eines Prozesses ist genau definiert, welche Ereignisse er im nächsten Schritt akzeptiert und welche nicht.

Sequenzen von Ereignissen, welche einen Prozess während der Ausführung leiten, werden *Traces* genannt. Ein Prozess kann unterschiedliche, durch die CSP-Semantik genau spezifizierte, Traces ausführen. CSP-Prozesse sind mathematische Abstraktionen der Interaktion des Systems mit dessen Umgebung.

Die Funktionsweise eine Lichtschalters lässt sich in CSP durch folgenden rekursiven Prozess definieren. `Switch = on -> off -> Switch`. Dieser akzeptiert zu Beginn das Ereignis `on`, danach nur `off`. Anschließend verhält er sich wieder wie `Switch`. Gültige Traces des Schalters sind $\langle\rangle$, $\langle$`on`$\rangle$, $\langle$`on`, `off`$\rangle$, $\langle$`on`, `off`, `on`$\rangle$, $\langle$`on`, `off`, `on`, `off`$\rangle$ und so weiter.

Das im Rahmen dieser Diplomarbeit erstellte Programm Jassda beobachtet den Ablauf von Java-Programmen. Durch das Java Debug Interface (JDI) erhält es Informationen über den aktuellen Zustand des Programms und erzeugt *Ereignisse*, sobald relevante *Methoden aufgerufen* oder *beendet* werden sowie bei Auftreten eines Ausnahmefehlers (englisch: *Exception*) und bei *Terminierung* des Programmes.

Mit der Spezifikations-Sprache CSP_{jassda} lassen sich Traces eines Systems beschreiben. Das System wird als Prozess spezifiziert, welcher seinerseits Subprozesse enthalten kann. Die Trace-Semantik des Prozesses definiert, welche Traces erlaubt sind.

CSP_{jassda} -Spezifikationen können interpretiert werden: Die operationelle Semantik eines Prozesses gibt an, wie dieser auf welche Ereignisse reagiert. Für jedes Ereignis, welches Jassda liefert, wird geprüft, ob eine Regel existiert, die den Prozess in eine Nachfolge-Prozess überführt. Falls eine derartige Regel existiert, wird der Nachfolge-Prozess zum aktuellen Prozess. Andernfalls wird ein Fehler angezeigt.

Die Syntax und Semantik von CSP_{jassda} werden nun formal eingeführt. Unterschiede zur CSP-Definition gemäß (Roscoe 1997), (Hoare 1985) werden erläutert.

3.2.2 Syntax

CSP wurde ursprünglich nicht als Sprache entwickelt, die von Computern verarbeitet werden kann. Aus diesem Grunde war die Syntax relativ komplex und es wurden Zeichen benutzt, die nicht zum ASCII-Zeichensatz gehörten. Als Forschungsgruppen begannen, Tools für CSP zu entwickeln, benötigten sie einen CSP Dialekt, der von Maschinen gelesen werden konnte. Mit der Entwicklung von FDR[1] durch *Formal Systems* entstand der CSP Dialekt CSP_M.

CSP_{jassda} ist ein Dialekt von CSP_M und speziell an die Verifikation von Traces von Java-Programmen angepasst. Zur Beschreibung erlaubter Traces stellt CSP_{jassda} eine Menge von Operatoren bereit.

Auf folgende Symbole wird im Verlauf dieser Arbeit zurückgegriffen.

- $a, b, event \in Events$: Menge aller *Ereignisse*, die den Aufruf oder das Ende einer Methode sowie das Auftreten einer Exception signalisieren.

[1] FDR ("Failures-Divergence Refinement") ist ein Model Checker für CSP

- $\mathtt{A}, \mathtt{B} \subseteq$ *Events*: *Alphabete* = Mengen von Ereignissen, an denen konkrete Prozesse teilnehmen können.
- $\checkmark \notin$ *Events*: Symbol, gelesen "Tick", zur Modellierung der Terminierung von Prozessen. Dieses Ereignis wird erzeugt, wenn die Java Virtual Machine, in der das untersuchte Programm abläuft, terminiert.
- $\alpha, \beta \in \mathit{Events} \cup \{\checkmark\} = \mathit{Events}^{\checkmark}$: Menge *aller* Ereignisse von Prozessen.
- $\mathtt{a}, \mathtt{b} \subseteq$ *Events*: *Ereignismengen* = Mengen von Ereignissen mit bestimmten Attributen.
- $\mathtt{M}, \mathtt{N} \subseteq \mathbb{P}(\mathit{Events})$: M und JassdaN sind disjunkte Teilmengen der Menge aller Events. Sie werden für die Definition von Variablen und quantisierenden Operatoren verwendet. (siehe Abschnitte 3.2.2.2 und 3.2.3.9)
- $\mathtt{X}, \mathtt{Y} \in$ *Idf*: Menge der (*Prozess-*)*Identifikatoren*, die als Namen für CSP-Prozesse dienen.

3.2.2.1 Ereignisse

Jassda informiert über Zustandsänderungen des untersuchten Programmes durch Erzeugen von Ereignissen. Die konkrete Ausprägung (Struktur) der Ereignisse ist durch Implementierung speziellen Java-Klassen anpassbar. Die minimale Struktur eines Ereignisses ist gegeben durch:

$$\mathit{event} = \begin{pmatrix} \mathit{type} \\ \mathit{jdi-interface} \\ \mathit{clocks-interface} \end{pmatrix}$$

Die Attribute des Events haben folgende Bedeutung:

type: Das type-Attribut zeigt die Aktion des untersuchten Programmes an, wodurch das Ereignis ausgelöst wurde. type kann folgende Werte annehmen:

- Aufruf (Beginn) einer Methode
- Verlassen (Ende) einer Methode
- Auftreten eines Ausnahmefehlers (*Exception*)
- Terminierung des untersuchten Programmes

jdi-interface: Referenz zum Java Debug Interface (JDI). Durch dieses Interface lassen sich Debug-Informationen über das untersuchte Programm erfragen (siehe auch Abschnitt 3.1.3.3).
Anmerkung: Alle Daten, die durch das JDI erfragbar oder berechenbar sind werden in dieser Arbeit implizit als Attribute des Ereignisses aufgefasst.

clocks-interface: Das Tool Jassda verwaltet ein Menge von Uhren. Über das `clocks-interface` lassen sich diese ansprechen.

Die Implementierung von Jassda bietet darüber hinaus folgende Attribute. Diese Attribute dienen dem bequemeren Zugriff auf häufig benutzte Informationen.

$$event = \begin{pmatrix} type \\ jdi-interface \\ clocks-interface \\ virtualmachine \\ thread \\ instance \\ method \\ parameter \\ returnvalue \end{pmatrix}$$

virtualmachine: Über das JDI können gleichzeitig mehrere Programme untersucht werden. Das Attribut `virtualmachine` liefert eine Referenz der Virtual Machine, welche das Ereignis erzeugt hat.

thread: `thread` bezeichnet den Thread.

instance: Java kann mehrere Instanzen einer Klasse erzeugen. Das Attribut `instance` bezeichnet die konkrete Instanz.

method: Die Methode, welche aufgerufen oder beendet wurde.

parameter: Es kann auf eine Liste der Methoden-Parameter zugegriffen werden.

returnvalue: Zeigt das Ereignis ein erfolgreiches Beenden einer Methode an, ist unter `returnvalue` der Rückgabewert zu finden.

3.2.2.2 Ereignismengen

Die von Jassda gelieferten Ereignisse sind sehr detailliert. Um den Ablauf eines Programmes zu spezifizieren, ist eine abstraktere Sicht sinnvoll, weil für die Spezifikation einer Trace häufig nicht alle Informationen von Interesse sind.

Ereignismengen ermöglichen, Ereignisse mit gemeinsamen Eigenschaften in einer Menge zu gruppieren. Es brauchen nur die Attribute angegeben zu werden, die für die Spezifikation von Interesse sind.

Beispiel: Angenommen, die Traces eines Schalters sind durch den Prozess `Switch1 = on -> off -> Switch1` definiert. Dieser Schalter wird in einem größeren System eingesetzt, dass einen weiteren Schalter enthält. Beide Schalter können unabhängig arbeiten. Nur die Trace des ersten Schalters sei hier von Interesse. Beim Schalten des ersten Schalters werden detaillierte Events erzeugt, die Informationen über die Stellung beider Schalter enthalten. Diese seien durch folgende Notation dargestellt:

$$event = \texttt{<state\_switch1>.<state\_switch2>}$$

Gültige Traces für den ersten Schalter im erweiterten System sind: $\langle\rangle$, $\langle$`on.on`$\rangle$, $\langle$`on.off`$\rangle$, $\langle$`on.on`$,$`off.on`$\rangle$, $\langle$`on.on`$,$`off.off`$\rangle$ und so weiter. Die Spezifikation der Traces des ersten Schalters ist mit den komplexeren Ereignissen schwieriger: Statt zwei unterschiedlicher Ereignisse (`on`, `off`) müssen vier betrachtet werden: `on.on`, `on.off`, `off.on` und `off.off`.
Das Konzept der Ereignismengen erlaubt, von den konkreten Ereignissen zu abstrahieren. Es werden nur die Informationen berücksichtigt, die zur Betrachtung des Traces des ersten Schalters notwendig sind. Ereignisse, die anzeigen, dass der erste Schalter den Wert `on` annimmt werden in der Menge `onSet` zusammengefasst, alle anderen in der Menge `offSet`. Die Traces des System mit zwei Schaltern werden durch `Switch1 = onSet -> offSet -> Switch1` beschrieben. □

Formale Definition von Ereignismengen Eine Ereignismenge `a` wird durch eine Abbildung f definiert, die angibt, ob ein konkretes Ereignis α in der Menge enthalten ist.

$$\texttt{a} := \{\alpha \in Events^{\checkmark} \mid f(\alpha) = true\}$$

wobei

$$\begin{aligned} f : Events^{\checkmark} &\rightarrow \mathbb{B} \\ \alpha &\mapsto f(\alpha) \end{aligned}$$

Die Abbildung f kann in Jassda durch spezielle Java-Klassen definiert werden. Es steht dem Anwender von Jassda frei, weitere Klassen für spezielle Anwendungen zu implementieren. (siehe Abschnitt 4.5.4.3)

Vordefinierte Ereignismengen Einige häufig benutzte Ereignismengen sind vordefiniert:

- alle Ereignisse – `any` := $Events^{\checkmark}$
- Methoden-Aufrufe – `begin` := $\{a \in Events \mid \text{type von a} = begin\}$
- Methoden-Enden – `end` := $\{a \in Events \mid \text{type von a} = end\}$
- Ausnahmefehler – `exception` := $\{a \in Events \mid \text{type von a} = exception\}$

Mengenoperatoren Vereinigungs- und Schitt-Operatoren können auf Ereignismengen angewendet werden

Vereinigung Die Symbole ”,” und ”+” werden als Vereinigungs-Operator interpretiert:

$$a,b := a+b := a \cup b$$

Schnitt Die Symbole ”.” und ”!” symbolisieren den Schnitt-Operator.

$$a.b := a!b := a \cap b$$

Beispiel: Seien `producer` und `init` Ereignismengen, die alle Ereignisse der Klasse *Producer* bzw. der Methode *init()* (einer beliebigen Klasse) spezifizieren, dann beschreibt

```
producer,init
```

bzw.

```
producer+init
```

eine Ereignismenge, die alle Ereignisse der Klasse *Producer* als auch alle Ereignisse der Methode *init()* enthält.

Die Ereignismenge

```
producer.init
```

bzw.

```
producer!init
```

beschreibt hingegen nur die Ereignisse die von der Methode *init()* aus der Klasse *Producer* erzeugt werden. □

Äquivalenzen: Seien `a1`, `a2` und `a3` Ereignismengen, `emptyset` die leere Menge und `fullset` die Menge aller Ereignisse. Die Äquivalenzen der Ereignismengen sind aus der Mengenlehre entnommen. Für die Vereinigung von Ereignismengen gelten folgende Äquivalenzen:

```
(a1,a2),a3  = a1,(a2,a3)
a1,a2       = a2,a1
a1,emptyset = a1
a1,fullset  = fullset
```

Für den Schnitt gilt:

```
(a1.a2).a3  = a1.(a2.a3)
a1.a2       = a2.a1
a1.emptyset = emptyset
a1.fullset  = a1
```

□

Bemerkung: Jassda nutzt diese Äquivalenzen, um komplexe Ereignismengen zu vereinfachen.

Variablen Das Konstrukt der Variablen ist an das Konzept der Kommunikation aus CSP_M angelehnt. Variablen sind Ereignismengen, die dazu dienen, Information aus Ereignissen zu speichern und an folgende Prozesse zu kommunizieren. Häufig sind nicht alle Informationen, die ein Ereignis liefert, für folgende Prozesse relevant. Bei der Wertzuweisung von Variablen wird daher vom konkreten Ereignis abstrahiert und nur die relevanten Informationen gespeichert.

Wertzuweisung von Variablen Die Wertzuweisung einer Variablen hat folgende Syntax:

```
a?x:M
```

dabei beschreibt `a` eine Ereignismenge, `x` eine Variable und `M` eine disjunkte Zerlegung der Menge aller *Events*, die durch die Abbildung g definiert ist:

$$\texttt{M} = \bigcup_{\alpha \in Events} \{g(\alpha)\}$$

g bildet ein Ereignis α in eine Ereignismenge, in der sich ähnliche Ereignisse befinden, ab.

$$g : Events \rightarrow \mathbb{P}(Events)$$

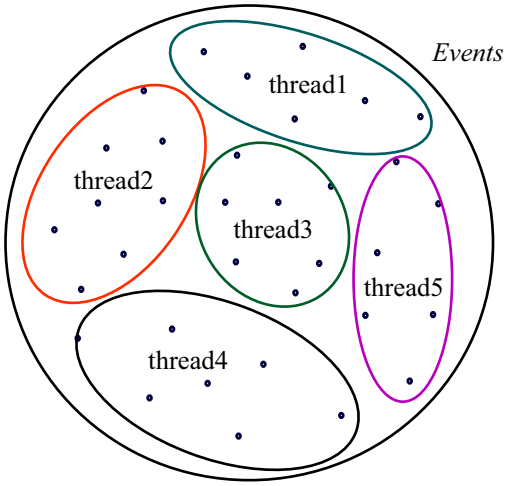

Abbildung 3.4: disjunkte Zerlegung (Unterscheidungsmerkmal: `thread`)

$$\begin{aligned} \alpha &\mapsto g(\alpha) \\ g(\alpha) &= \{\beta \in \mathtt{a} \subseteq \mathit{Events} \mid \alpha \in \mathtt{a} \\ &\qquad \wedge\, \beta \text{ hat dieselbe Eigenschaft wie } \alpha\} \end{aligned}$$

Durch geeignete Wahl der disjunkten Zerlegung `M` kann festgelegt werden, welche Informationen des Ereignisses α gespeichert werden.

Die Abbildungen 3.4 und 3.5 veranschaulichen die disjunkte Zerlegung. In Abbildung 3.4 sind alle Ereignisse, die von einem bestimmten Thread stammen, in einer Menge zusammengefasst. Mit dieser Zerlegung wird in der Variablen die Information `thread` des Ereignisses gespeichert. Abbildung 3.5 zeigt hingegen eine disjunkte Zerlegung auf Basis der Instanzen. In diesem Fall speichert die Variable die Information `instance` des Ereignisses. Die Abbildung g wird durch spezielle Java-Klassen realisiert. Es steht dem Benutzer frei, für besondere Ansprüche eigene Klassen zu implementieren. Der obige Ausdruck der Wertzuweisung wird in CSP_{jassda} syntaktisch als Ereignismenge interpretiert.

Die Wertzuweisung geschieht wie folgt:

1. Es wird geprüft, ob ein Ereignis $\alpha \in \mathit{Events}$ Element der Ereignismenge `a` ist.
2. Ist $\alpha \notin \mathtt{a}$, dann wird keine Wertzuweisung durchgeführt und ein Fehler ausgegeben.
3. Gilt $\alpha \in \mathtt{a}$, wird die Ereignismenge $g(\alpha)$ aus dem Ereignis α berechnet und der Variablen `x` zugewiesen.

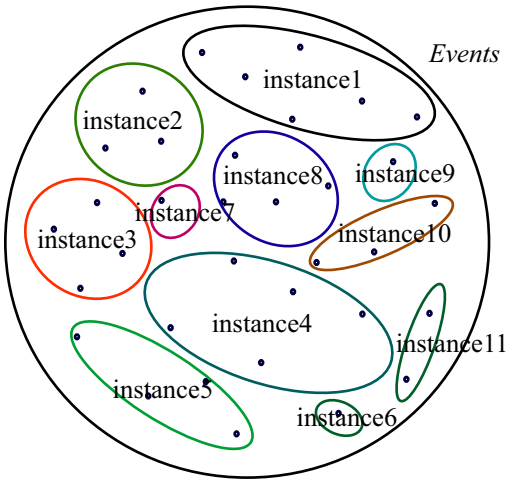

Abbildung 3.5: disjunkte Zerlegung (Unterscheidungsmerkmal: `instance`)

Die Abbildung `g` dient der Abstraktion vom konkreten Ereignis. Die Ereignismenge `x` enthält alle Informationen des Ereignisses α, die durch g ausgewählt werden.

Nutzung von Variablen Sobald einer Variablen ein Wert zugeordnet wurde, kann sie genauso eingesetzt werden wie statische Ereignismengen. Die Mengenoperatoren Schnitt und Vereinigung unterscheiden nicht zwischen variablen und statischen Ereignismengen. Um in der Spezifikation syntaktisch zu unterstreichen, dass eine Schnittmenge mit einer variablen Ereignismenge gebildet wird, ist das "!" dem "." vorzuziehen. Auf diese Weise ähnelt die Notation eher der Kommunikation von CSP_M.

Beispiel: Sei `producer` eine Ereignismenge, die alle Ereignisse der Klasse "Producer" und `produce` eine Ereignismenge, die alle Ereignisse der Methode "produce()" enthält. M sei definiert durch die Abbildung $g_{instance}$ mit

$$g_{instance}(\alpha) = \{\beta \in \mathtt{a} \subseteq \mathit{Events} \mid \alpha \in \mathtt{a} \\ \wedge\, \beta \text{ stammt von derselben } \mathit{Instanz} \text{ wie } \alpha\}$$

Weiterhin wird angenommen, das mehrere Instanzen der Klasse "Producer" existieren. Es wird gewünscht, die Trace eines Programmes pro Objekt-Instanz zu betrachten.

```
producer.init.begin?instance:M
```

```
-> producer.init.end!instance
-> producer.produce.begin!instance
-> ANY
```

Für jede Instanz muss `init()` aufgerufen und beendet. Anschließend wird die Methode 'produce()' aufrufen. □

3.2.2.3 Prozesse

Mit den obigen Vorbemerkungen kann die Menge *Spec* der CSP_{jassda} -Prozesse durch folgende Syntaxregeln definiert werden. Dabei seien $\mathtt{P}, \mathtt{Q} \in$ *Spec*, $\mathtt{A}, \mathtt{B} \subseteq$ *Events* (Alphabete, siehe Abschnitt 3.2.2.4) und $\mathtt{a}, \mathtt{x} \subseteq$ *Events*.

```
P ::= STOP
    | TERM
    | ANY[A]
    | a -> P
    | P ; Q
    | P [] Q
    | P [A||B] Q
    | X
```

Die einzelnen Konstrukte bedeuten:

Deadlock (`STOP`)
Der Prozess ist zu keinen Aktionen bereit.

Terminierung (`TERM`)
Der Prozess ist nur noch zur ✓-Aktion (Terminierung) bereit.

Bemerkung: `TERM` entspricht dem `SKIP`-Prozess.

Don't care (`ANY[A]`)
Beliebige Aktionen werden akzeptiert.

Bemerkung: `ANY[A]` entspricht dem `Chaos`-Prozess.

Präfix-Operator (a -> P)
Der Prozess reagiert auf ein Ereignis aus der Ereignisgruppe a und verhält sich danach wie P. Diese Notation stellt eine Abstraktion des in CSP_M üblichen Präfix-Operators dar. a entspricht keinem konkreten Ereignis, sondern einer Gruppe von Ereignissen. Formal ist dies eine Kurzschreibweise für den Mengenpräfix-Operator ([]α:A @ α-> P(α)) wobei A = a) aus CSP_M.

Sequenz-Operator (P ; Q)
Sobald P nur noch das Terminierungs-Ereignis akzeptieren kann, wird Q gestartet.

Auswahl (P [] Q)
Entweder wird P ausgeführt oder Q.

Quantifizierende Auswahl ([]x:M @ P(x)) Für jedes x ∈ M existiert ein parametrisierter Prozess P(x). Die quantifizierende Auswahl kann ein Ereignis verarbeiten, wenn mindestens einer der parametrisierten Prozesse das Ereignis verarbeiten kann.

Parallel-Komposition (P [A||B] Q)
Die Prozesse P und Q werden parallel ausgeführt. Sie synchronisieren sich auf dem Schnitt der *Synchronisationsalphabete* A und B. Falls A und B nicht angegeben werden, wird über dem Schnitt der Alphabete von P und Q ($\alpha((P)) \cap \alpha(Q)$) synchronisiert. (Das Alphabet eines Prozesses wird in Abschnitt 3.2.2.4 definiert)

Quantifizierende Parallel-Komposition (||x:M @ P(x))
Für jedes x ∈ M existiert ein parametrisierter Prozess P(x). Diese Prozesse werden parallel ausgeführt.

Überwachter Aufruf (X)
Prozesse können mit einem Identifikator bezeichnet werden. Dieser kann benutzt werden, um den benannten Prozess aufzurufen. Jedem Identifikator darf nur ein Prozess zugeordnet werden.

3.2.2.4 Alphabete

Das Alphabet A = α(P) eines Prozesses P kann induktiv gemäß folgender Definitionen berechnet werden:

$$\alpha(\mathsf{STOP}) = \{\}$$
$$\alpha(\mathsf{TERM}) = \{\checkmark\}$$

$$\begin{aligned}
\alpha(\texttt{ANY[A]}) &= \texttt{A} \\
\alpha(\texttt{a->P}) &= \texttt{a} \cup \alpha(\texttt{P}) \\
\alpha(\texttt{P ; Q}) &= \alpha(\texttt{P}) \cup \alpha(\texttt{Q}) \\
\alpha(\texttt{P [] Q}) &= \alpha(\texttt{P}) \cup \alpha(\texttt{Q}) \\
\alpha(\texttt{[]x:M @ P(x)}) &= \bigcup_{\texttt{x} \in \texttt{M}} \alpha(\texttt{P(x)}) \\
\alpha(\texttt{P [A||B] Q}) &= \texttt{A} \cup \texttt{B} \\
\alpha(\texttt{||x:M @ P(x)}) &= \bigcup_{\texttt{x} \in \texttt{M}} \alpha(\texttt{P(x)})
\end{aligned}$$

3.2.3 Operationelle Semantik

Die operationelle Semantik ermöglicht es, eine Sprache zu interpretieren - schrittweise ein in dieser Sprache geschriebenes Programm auszuführen. CSP_{jassda} bezieht sich auf die Ausführung von Ereignissen. Die operationelle Semantik beschreibt daher, wann welche Ereignisse verarbeitet werden können.

Genauer gesagt bedeutet 'operationell', dass für jeden CSP_{jassda} -Prozess die Zustände und Transitionen einer abstrakten Maschine beschrieben werden (Plotkin 1981). Als abstrakte Maschinen werden hier Automaten der Form

$$A = (A, S, \rightarrow, s_0)$$

benutzt, wobei $A \subseteq$ *Events* ein *Alphabet* ist, $S \in$ *Spec* eine Menge von *Zuständen*, $s_0 \in S$ der *Anfangszustand* und

$$\rightarrow \subseteq S \times (A \cup \{\checkmark\}) \times S$$

die Transitionsrelation. Die Zustände der abstrakten Maschine repräsentieren CSP_{jassda} -Prozesse.

Die *Transitionen* (die Elemente der Transitionsrelation) haben folgende Form

$$\texttt{P} \xrightarrow{\alpha} \texttt{Q}$$

wobei $\texttt{P}, \texttt{Q} \in$ *Spec* Prozesse sind und $\alpha \in$ *Events*$^{\checkmark}$ ein Ereignis ist. Eine solche Relation beschreibt, dass der Prozess P anfangs an dem Ereignis α teilnehmen kann und sich dann wie der Prozess Q verhält. (Olderog 2001)

Die Semantik von CSP_{jassda} wird durch *Transitionsregeln* induktiv über den syntaktischen Aufbau der CSP_{jassda} -Prozesse beschrieben.

Transitionsregeln Eine Transitionsregel hat folgende allgemeine Form

$$\frac{\begin{array}{c} T_1 \\ \vdots \\ T_n \end{array}}{T} \quad [\ \textit{Bedingung}\ ]$$

wobei $n \geq 0$ und $T_1, \ldots, T_n$, T Schemata für Transitionen sind. Dies bedeutet anschaulich: Wenn die Prämissen (die Transitionen $T_1, \ldots, T_n$) die Bedingung *Bedingung* erfüllen, dann gilt auch die Konklusion (die Transition T).

3.2.3.1 Deadlock

Der Prozess ist unter keinen Umständen zu einer Aktion bereit.

3.2.3.2 Terminierung

Dieser Prozess zeigt die erfolgreiche Terminierung eines Programmes an. Genauer gesagt, beschreibt er den Zustand der Java Virtual Machine zu dem Zeitpunkt, an dem das untersuchte Programm terminiert ist, die JVM jedoch noch nicht. Es wird nur noch das ✓-Ereignis akzeptiert, welches die Terminierung der JVM repräsentiert.

$$\frac{}{\mathsf{TERM} \xrightarrow{\checkmark} \mathsf{STOP}}$$

3.2.3.3 Don't care

Jedes Ereignis aus dem Alphabet des ANY-Operators überführt ihn in sich selbst. Eine Ausnahme stellt das Terminierungs-Ereignis ✓dar: Nach einem ✓ können keine weiteren Ereignisse verarbeitet werden.

$$\frac{}{\mathsf{ANY[A]} \xrightarrow{\alpha} \mathsf{ANY[A]}} \quad [\ \alpha \in \mathsf{A} \subseteq \textit{Events}\ ]$$

$$\frac{}{\mathsf{ANY[A]} \xrightarrow{\checkmark} \mathsf{STOP}}$$

3.2.3.4 Präfix-Operator

Dieser Prozess verarbeitet nur Ereignisse α, die in der Ereignismenge a enthalten sind und verhält sich danach wie P.

$$\frac{}{(\mathsf{a}\ \texttt{->}\ \mathsf{P}) \xrightarrow{\alpha} \mathsf{P}} \quad [\ \alpha \in \mathsf{a} \subseteq \textit{Events}\ ]$$

Beispiel: Eine gültige Trace eines Erzeugers sei dadurch gegeben, dass dieser zunächst initialisiert werden muss und erstes danach mit der Produktion beginnt.

```
producer.init.begin
 -> producer.init.end
 -> producer.produce.begin
 -> ANY
```

□

3.2.3.5 Sequenz-Operator

Die sequentielle Komposition `P ; Q` beschreibt, dass der Prozess `Q` gestartet wird, sobald `P` das Terminierungs-Ereignis (✓) und `Q` das aktuelle Ereignis α akzeptieren. Bei Ereignissen α ($\neq \checkmark$), die von `P` verarbeitet werden können, wird `Q` nicht gestartet.

$$\frac{\mathtt{P} \xrightarrow{\alpha} \mathtt{P'}}{\mathtt{P;Q} \xrightarrow{\alpha} \mathtt{P';Q}} \quad [\ \checkmark \neq \alpha \in \mathit{Events}\ ]$$

$$\frac{\begin{array}{c}\mathtt{P} \xrightarrow{\checkmark} \mathtt{P'} \\ \mathtt{Q} \xrightarrow{\alpha} \mathtt{Q'}\end{array}}{\mathtt{P;Q} \xrightarrow{\checkmark} \mathtt{Q'}} \quad [\ \alpha \in \mathit{Events}^{\checkmark}\ ]$$

3.2.3.6 Auswahl

Der Auswahl-Operator $[]_{CSP_M}$ entscheidet gemäß der Definition von CSP_M anhand des ersten Ereignisses, welcher der alternativen Prozesse `P` oder `Q` ausgeführt wird. Können sowohl `P` als auch `Q` ein Ereignis $\alpha \in \mathit{Events}^{\checkmark}$ verarbeiten, wird nichtdeterministisch einer der Prozesse ausgewählt.

$$\frac{\mathtt{P} \xrightarrow{\alpha} \mathtt{P'}}{\mathtt{P}\ []_{CSP_M}\ \mathtt{Q} \xrightarrow{\alpha} \mathtt{P'}} \quad [\ \alpha \in \mathit{Events}^{\checkmark}\ ]$$

$$\frac{\mathtt{Q} \xrightarrow{\alpha} \mathtt{Q'}}{\mathtt{P}\ []_{CSP_M}\ \mathtt{Q} \xrightarrow{\alpha} \mathtt{Q'}} \quad [\ \alpha \in \mathit{Events}^{\checkmark}\ ]$$

Beispiel: Zwei Verbraucher (`consumer1`, `consumer2`) können ein Element (`element`) verbrauchen. Es ist nicht festgelegt, welcher Verbraucher das Element konsumiert (`consume`).

```
consumer1.consume.element.begin
 -> consumer1.consume.element.end
 -> ANY
[]CSP_M
consumer2.consume.element.begin
 -> consumer2.consume.element.end
 -> ANY
```

□

Verzögerte Auswahl Der Nichtdeterminismus der Auswahl von CSP_M ist für die Validierung von Java-Programmen ungeeignet. Daher wird in CSP_{jassda} die Entscheidung bei der Auswahl so lange verzögert, bis sie eindeutig anhand eines Ereignisses entscheidbar ist.

$$\frac{\mathtt{P} \xrightarrow{\alpha} \mathtt{P'}}{\mathtt{P\ []\ Q} \xrightarrow{\alpha} \mathtt{P'}} \quad [\ \alpha \in \mathit{Events}^{\checkmark} \text{ und } \nexists \mathtt{Q'} : \mathtt{Q} \xrightarrow{\alpha} \mathtt{Q'}\ ]$$

$$\frac{\mathtt{Q} \xrightarrow{\alpha} \mathtt{Q'}}{\mathtt{P\ []\ Q} \xrightarrow{\alpha} \mathtt{Q'}} \quad [\ \alpha \in \mathit{Events}^{\checkmark} \text{ und } \nexists \mathtt{P'} : \mathtt{P} \xrightarrow{\alpha} \mathtt{P'}\ ]$$

$$\frac{\begin{array}{c}\mathtt{P} \xrightarrow{\alpha} \mathtt{P'} \\ \mathtt{Q} \xrightarrow{\alpha} \mathtt{Q'}\end{array}}{\mathtt{P\ []\ Q} \xrightarrow{\alpha} \mathtt{P'\ []\ Q'}} \quad [\ \alpha \in \mathit{Events}^{\checkmark}\ ]$$

Die *verzögerte Auswahl* `[]` unterscheidet sich operationell von der Auswahl $[]_{CSP_M}$ in CSP_M. Zur Validierung von Abläufen eines Programmes wird jedoch die abstraktere *Trace-Semantik* genutzt. In der Trace-Semantik sind *Auswahl* und *verzögerte Auswahl* äquivalent, wie in Abschnitt 3.2.4.6 bewiesen wird.

3.2.3.7 Parallel-Komposition

Die Parallel-Komposition ermöglicht die gleichzeitige Überprüfung mehrerer Prozesse. Die Prozesse verfügen jeweils über ein Synchronisationsalphabet, welches entweder implizit gemäß Abschnitt 3.2.2.4 berechnet wird oder explizit angegeben werden kann. Sie synchronisieren sich auf Ereignissen α, die in der Schnittmenge der Synchronisationsalphabete $\mathtt{A} \cap \mathtt{B}$ enthalten sind.

$$\frac{P \xrightarrow{\alpha} P'}{P\ [A||B]\ Q \xrightarrow{\alpha} P'[A||B]\ Q} \quad [\,\alpha \in A \backslash B\,]$$

$$\frac{Q \xrightarrow{\alpha} Q'}{P\ [A||B]\ Q \xrightarrow{\alpha} P\ [A||B]\ Q'} \quad [\,\alpha \in B \backslash A\,]$$

$$\frac{\begin{array}{c} P \xrightarrow{\alpha} P' \\ Q \xrightarrow{\alpha} Q' \end{array}}{P\ [A||B]\ Q \xrightarrow{\alpha} P'\ [A||B]\ Q'} \quad [\,\alpha \in (A \cap B)^{\checkmark}\,]$$

Beispiel: Erzeuger und Verbraucher sind jeweils als Threads implementiert. Sie dürfen mit der Arbeit erst beginnen, wenn ein Start-Zeichen (`sync`) gegeben wurde.

```
producer.init.begin
 -> producer.init.end
 -> sync
 -> producer.run.begin
 -> ANY
||
consumer.init.begin
 -> consumer.init.end
 -> sync
 -> consumer.run.begin
 -> ANY
```

□

3.2.3.8 Überwachter Aufruf

Durch den Aufruf eines Identifikators können aus einem Prozess andere Prozesse aufgerufen werden. Dem referenzierten Prozess können Ereignismengen als Parameter übergeben werden. Überwachte Rekursion ist erlaubt. Überwachte Rekursion fordert, dass zwischen zwei Aufrufen mindestens ein Ereignis verarbeitet werden muss.

$$\frac{P \xrightarrow{\alpha} P'}{X \xrightarrow{\alpha} P'} \quad [\,X \text{ ist deklariert durch } P\,]$$

Beispiel: Ein Erzeuger (x) produziert abwechselnd unendlich oft die Elemente `elementa` und `elementb`.

```
P(x){x.produce.elementa.begin
     -> x.produce.elementa.end
     -> x.produce.elementb.begin
     -> x.produce.elementb.end
     -> P(x)
    }
```

□

3.2.3.9 Quantifizierende Operatoren

Java ermöglicht, zur Laufzeit Objekte zu instanziieren und Threads zu erzeugen. Die konkrete Ausprägung und Anzahl der Objekte und Threads ist häufig erst zur Laufzeit bekannt. Zur Modellierung dieses Verhaltens sind Ausdrücke wie "es existiert ein Thread, der ..." oder "für alle Instanzen der Klasse *Producer* gilt: ..." hilfreich.
CSP_{jassda} stellt für diesen Zweck quantifizierende Operatoren bereit, welche sich an die generalisierte Form der Auswahl- und Parallel-Operatoren aus CSP_M anlehnen. Es werden nun die generalisierten Operatoren hergeleitet und anschließend die Erweiterungen vorgestellt, die zu den quantisierenden Operatoren in CSP_{jassda} führen.

Generalisierte Auswahl Die generalisierte Auswahl $[]_{j\in I}$ geht durch wiederholte Ausführung aus der binären Auswahl (siehe Abschnitt 3.2.3.6) hervor.

$$[]_{j\in I} = \mathsf{P}_1 [] \mathsf{P}_2 [] \ldots [] \mathsf{P}_n \quad [\, I = \{1,2,\ldots,n\} \,]$$

Damit gilt:

$$\frac{\mathsf{P}_j \xrightarrow{\alpha} \mathsf{P}'}{[]_{i\in I} \mathsf{P}_i \xrightarrow{\alpha} \mathsf{P}'} \quad [\, j \in I, \alpha \in \mathit{Events}^{\checkmark} \,]$$

Generalisierter Parallel-Operator Der generalisierte Parallel-Operator $||_{A_i}^{i\in I}$ kann durch wiederholte Ausführung des binären Parallel-Operators definiert werden. Enthält die Indexmenge I nur zwei Elemente i_1 und i_2, gilt:

$$||_{A_i}^{i\in\{i_1,i_2\}} \mathsf{P}_{i_1} = \mathsf{P}_{i_1} \;\; [\;\; A_{i_1} \;\; || \;\; A_{i_2} \;\;] \;\; \mathsf{P}_{i_2}$$

Sei der Parallel-Operator mit n Elementen gegeben, so kann eine Definition für den n+1 elementigen Operator angegeben werden. Der n elementige Operator verarbeitet alle Ereignisse, die in der Vereinigung der Synchronisationsalphabete vorkommen ($\alpha(||_{A_i}^{i\in I} \mathtt{P}_i) = \bigcup_{i\in I} A_i$). Wird eine weitere Komponente $\mathtt{P}_j$ ($j \notin I$) mit dem Synchronisationsalphabet A_j hinzugefügt, resultiert dies in $||_{A_i}^{i\in I\cup\{j\}} \mathtt{P}_i$, wobei

$$||_{A_i}^{i\in I\cup\{j\}} \mathtt{P}_i = (||_{A_i}^{i\in I}) \ \mathtt{[} \ \cup_{i\in I} A_i \ \mathtt{||} \ A_j \ \mathtt{]} \ \mathtt{P}_j.$$

Anpassung an Java Mit der obigen Generalisierung sind Auswahl und Parallel-Operator so erweitert worden, dass Aussagen über eine beliebige Anzahl von Prozessen getroffen werden können. Wie wird jedoch modelliert, dass zum Beispiel alle Threads zunächst initialisiert werden müssen und erst dann starten dürfen?

```
∀ thread ∈ AllThreads :
  thread.init.begin
   -> thread.init.end
   -> thread.run.begin
   -> ANY
```

Für diesen Zweck erhält die Indexmenge I aus der Herleitung der Generalisierung in CSP_{jassda} eine besondere Bedeutung: Sie repräsentiert eine *disjunkte* Zerlegung (wie bereits in Abschnitt 3.2.2.2 eingeführt) der Menge aller Ereignisse und wird nachfolgend `M` genannt. Die Elemente der Zerlegung `M` sind paarweise disjunkte Ereignismengen, die Ereignisse mit gemeinsamen Eigenschaften zusammenfassen, und repräsentieren die einzelnen Indizes.

Beispielsweise könnte jeder einzelne Java-Thread eines Programmes durch genau eine Ereignismenge repräsentiert werden (siehe Abbildung 3.4). Eine Ereignismenge würde somit aus allen Ereignissen bestehen, die der jeweilige Thread erzeugen kann.
Die Menge, die diese Ereignismengen als Elemente enthält, beschreibt also die disjunkte Zerlegung aller möglichen Ereignisse eines Programmes und wird mit `M` bezeichnet. Die Zerlegung `M` entspricht der Zerlegung, die bei der Wertzuweisung von Variablen definiert wurde (siehe 3.2.2.2).

Mit Hilfe dieser neuen Interpretation der Indexmenge ergeben sich die quantifizierende Auswahl

```
[]x:M @ P(x)
```

und die quantifizierende Parallel Komposition

```
||x:M @ P(x)
```

wobei $\mathtt{x} \subseteq \mathit{Events}$ eine Ereignismenge, $\mathtt{M} \in \mathbb{P}(\mathit{Events})$ eine disjunkte Zerlegung und P ein Prozess, dem als Parameter die Ereignismenge x übergeben wird. Die Synchronisationsalphabete bei der quantifizierenden Parallel-Komposition werden implizit berechnet.

Beispiel: Sei die disjunkte Zerlegung M durch die Abbildung g_{thread} mit

$$\begin{aligned} g_{thread} &: \mathit{Events} \to \mathbb{P}(\mathit{Events}) \\ &\alpha \mapsto g_{thread}(\alpha) \\ g_{thread}(\alpha) &= \{\beta \in \mathtt{x} \subseteq \mathit{Events} \mid \alpha \in \mathtt{x} \\ &\qquad \wedge\, \beta \text{ hat denselben } \textbf{Thread} \text{ wie } \alpha\} \end{aligned}$$

gegeben (Visualisierung siehe Abb. 3.4), dann kann mit

```
||thread:M @ thread.init.begin
 -> thread.init.end
 -> thread.run.begin
 -> ANY
```

spezifiziert werden, dass in jedem einzelnen *Thread* zunächst die 'init()'-Methode erfolgreich abgearbeitet werden muss und danach die 'run()'-Methode ausgeführt wird.
Ist die disjunkte Zerlegung M hingegen durch $g_{instance}$

$$\begin{aligned} g_{instance} &: \mathit{Events} \to \mathbb{P}(\mathit{Events}) \\ &\alpha \mapsto g_{instance}(\alpha) \\ g_{instance}(\alpha) &= \{\beta \in \mathtt{x} \subseteq \mathit{Events} \mid \alpha \in \mathtt{x} \\ &\qquad \wedge\, \forall\, \mathtt{y} \subseteq \mathit{Events} : \mathtt{x} \neq \mathtt{y} \Leftrightarrow \mathtt{x} \cap \mathtt{y} = \varnothing \\ &\qquad \wedge\, \beta \text{ stammt von derselben } \textbf{Instanz} \text{ wie } \alpha\} \end{aligned}$$

definiert (Visualisierung siehe 3.5), wird über Objekt-Instanzen quantifiziert werden:

```
||instance:M @ instance.init.begin
 -> instance.init.end
 -> instance.run.begin
 -> ANY
```

In diesem Fall muss für jedes einzelne *Objekt* zunächst die 'init()'-Methode erfolgreich ausgeführt werden. Anschließend ist die 'run()'-Methode aufzurufen. □

3.2.4 Trace-Semantik

Die operationelle Semantik beschreibt sehr detailliert, wie Ereignisse von den einzelnen CSP_{jassda} Operatoren verarbeitet werden. Eine etwas abstraktere Sicht auf das Verhalten eines Programmes bietet die Trace-Semantik. Unter einer *Trace* wird einen endliche Sequenz von Ereignissen verstanden. Zur Beschreibung von Traces werden folgende Symbole benutzt:

- Traces := Menge der Traces, die aus Ereignisse aus $Events^{\checkmark}$ bestehen
- Spec := Menge der CSP_{jassda} -Prozesse
- $\langle\,\rangle$:= leere Trace
- $\langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle$:= Trace bestehend aus den Ereignissen $\alpha_1, \alpha_2, \ldots, \alpha_n$
- $\mathtt{P} =_T \mathtt{Q}$ beschreibt die Äquivalenz der Prozesse P und Q bezüglich der Trace-Semantik.
- $s \frown t$ beschreibt die Konkatenation der Traces s und t
- $s \restriction \mathtt{a}$ ist die Projektion der Trace s auf die Ereignismenge a. Sei beispielsweise $\mathtt{a} = \{\alpha_1, \alpha_3\}$ und $s = \langle \alpha_2, \alpha_3, \alpha_1, \alpha_4 \rangle$, dann ist $s \restriction \mathtt{a} = \langle \alpha_3, \alpha_1 \rangle$.

Die *erweiterte Transitionsrelation* $\mathtt{P} \stackrel{s}{\Longrightarrow} \mathtt{Q}$ beschreibt, dass eine Sequenz s von Transitionen (gemäß der operationellen Semantik) existiert, dessen Start-Prozess P und End-Prozess Q ist. Formal bedeutet dies:

$$\Longrightarrow \subseteq \mathit{Spec} \times \mathit{Traces} \times \mathit{Spec}$$

wobei:

$$\forall \mathtt{P} \in \mathit{Spec} : (\mathtt{P}, \langle\,\rangle, \mathtt{P}) \in \Longrightarrow$$

und

$$\begin{aligned}&\forall \mathtt{P}, \mathtt{Q} \in \mathit{Spec}, \alpha \in \mathit{Events}^{\checkmark}, s \in \mathit{Traces} : \\ &\quad (\mathtt{P}, \langle \alpha \rangle \frown s, \mathtt{Q}) \in \Longrightarrow \\ &\quad \Leftrightarrow \\ &\quad (\exists \mathtt{P}' \in \mathit{Spec} : \mathtt{P} \stackrel{\alpha}{\rightarrow} \mathtt{P}' \wedge (\mathtt{P}', s, \mathtt{Q}) \in \Longrightarrow)\end{aligned}$$

Weiterhin gilt:

$$\mathtt{P} \stackrel{s}{\Longrightarrow} \mathtt{Q} :\Leftrightarrow (\mathtt{P}, s, \mathtt{Q}) \in \Longrightarrow$$

Unter Verwendung der erweiterten Transitionsrelation lässt sich die Trace-Semantik *traces* von CSP_{jassda} -Prozessen definieren:

$$traces : Spec \rightarrow \mathbb{P}(Traces)$$

wobei

$$\forall \mathtt{P} \in Spec :$$
$$traces(\mathtt{P}) = \{s \in Traces \mid \exists \mathtt{Q} \in Spec : \mathtt{P} \stackrel{s}{\Longrightarrow} \mathtt{Q}\}$$

Die Trace-Semantik der einzelnen CSP_{jassda} -Operatoren wird nun vorgestellt.

3.2.4.1 Deadlock

Nur die leere Trace kann mit dem `STOP`-Prozess gebildet werden.

$$traces(\mathtt{STOP}) = \{\langle\rangle\}$$

3.2.4.2 Terminierung

Entweder es wurde noch kein Ereignis ($\alpha \neq \checkmark$) beobachte und die Trace ist leer oder das Programm wurde terminiert und die Trace besteht aus dem Terminierungs-Ereignis $\checkmark$.

$$traces(\mathtt{TERM}) = \{\langle\rangle, \langle\checkmark\rangle\}$$

3.2.4.3 Don't care

Der `ANY`-Prozess bildet in der Trace-Semantik die Menge aller Trace die mit einem $\checkmark$-Ereignis abgeschlossen werden.

$$traces(\mathtt{ANY}) = seq(Events) \frown \{\checkmark\}$$

3.2.4.4 Präfix-Operator

Die Traces des Präfix-Operators werden durch zwei Fälle charakteriert: Entweder es ist bisher kein Ereignis $\alpha \in \mathtt{a}$ aufgetreten, oder es ist ein $\alpha \in \mathtt{a}$ Ereignis aufgetreten und der Rest des Traces ergibt sich aus dem Prozess `P`.

$$\begin{aligned} traces(\mathtt{a\ ->\ P}) = & \{\langle\rangle\} \\ & \cup \\ & \{\langle\alpha\rangle\} \frown traces(\mathtt{P}) \end{aligned}$$

In der Trace-Semantik von CSP_{jassda} besteht folgende Äquivalenz bezüglich des Präfix-Operators:

Äquivalenz: Sei die Ereignismenge $\mathtt{a} = \varnothing = \{\}$ die leere Menge, dann gilt:

$$\mathtt{a \ -> \ P} =_T \mathtt{STOP}$$

□

3.2.4.5 Sequenz-Operator

Die sequentielle Komposition `P ; Q` verhält sich wie `P` bis `P` terminieren könnte (= das Terminierungs-Ereignis ✓verarbeiten könnte) und `Q` das aktuelle Ereignis verarbeiten kann. Ist dies der Fall, wird die Ablaufkontrolle an `Q` übergeben.

$$\begin{aligned} \mathit{traces}(\mathtt{P \ ; \ Q}) = & \{s \in \mathit{traces}(\mathtt{P}) \mid \checkmark \text{ kommt nicht in } s \text{ vor}\} \\ & \cup \\ & \{s \frown t \in \mathit{Traces} \mid s \frown \langle\checkmark\rangle \in \mathit{traces}(\mathtt{P}) \wedge t \in \mathit{traces}(\mathtt{Q})\} \end{aligned}$$

Äquivalenzen: Für die sequentielle Komposition gelten folgende Äquivalenzen:

$$\begin{aligned} \mathtt{(P1;P2);P3} &=_T \mathtt{P1;(P2;P3)} \\ \mathtt{TERM;P} &=_T \mathtt{P} \\ \mathtt{P;TERM} &=_T \mathtt{P} \\ \mathtt{(a->P);Q} &=_T \mathtt{a->(P;Q)} \\ \mathtt{STOP;P} &=_T \mathtt{STOP} \end{aligned}$$

□

3.2.4.6 Auswahl

Nach der operationelle Semantik des Ausdrucks $\mathtt{P} \ []_{CSP_M} \ \mathtt{Q}$ wird entweder `P` oder `Q` ausgeführt. Die möglichen Traces ergeben sich also entweder aus der Menge der Traces von `P` oder `Q`. laut CSP_M-Trace-Semantik von $[]_{CSP_M}$ gilt:

$$\begin{aligned} \mathit{traces}(\mathtt{P} \ []_{CSP_M} \ \mathtt{Q}) = & \mathit{traces}(\mathtt{P}) \\ & \cup \\ & \mathit{traces}(\mathtt{Q}) \end{aligned}$$

Im nachfolgenden Beweis wird gezeigt, dass der in CSP_{jassda} betrachtete Operator `[]` der verzögerten Auswahl äquivalent zum Auswahl Operator $[]_{CSP_M}$ ist:

$$\mathtt{P} \ []_{CSP_M} \ \mathtt{Q} =_T \mathtt{P \ [] \ Q}$$

Beweis: Zu zeigen:

$$\begin{aligned} traces(\mathtt{P\ []\ Q}) = & \ traces(\mathtt{P}) \\ & \cup \\ & \ traces(\mathtt{Q}) \end{aligned}$$

Der Beweis wird durch Induktion über die Länge der Traces geführt:

1. **Induktionsanfang**: Die leere Trace ist ein Element der Menge der Traces von `P [] Q` als auch von `P` und `Q`.

   $$\langle\rangle \in traces(\mathtt{P\ []\ Q}), traces(\mathtt{P}), traces(\mathtt{Q})$$

2. **Induktionsannahme** Es wird angenommen, dass für alle Traces $s \in Traces$ mit der Länge n gilt:

   $$s \in traces(\mathtt{P[]Q}) \Leftrightarrow s \in traces(\mathtt{P}) \cup traces(\mathtt{Q})$$

3. **Induktionsschritt** Sei

   $$t = \langle\alpha\rangle \frown s$$

   eine Trace der Länge n+1.

   - Sei $t \in traces(\mathtt{P\ []\ Q})$. Zu zeigen: $t \in traces(\mathtt{P}) \cup traces(\mathtt{Q})$.
     Nach Definition der Trace-Semantik gilt:

     $$t \in traces(\mathtt{P\ []\ Q}) \Rightarrow \exists \mathtt{R} \in Spec : \mathtt{P\ []\ Q} \stackrel{t}{\Longrightarrow} \mathtt{R}$$

     Folgende Fälle sind zu unterscheiden:

     (a) $\nexists \mathtt{Q'} : \mathtt{Q} \stackrel{\alpha}{\rightarrow} \mathtt{Q'}$

     $$\begin{aligned} & \Rightarrow \exists \mathtt{P'} \in Spec : \mathtt{P\ []\ Q} \stackrel{\alpha}{\rightarrow} \mathtt{P'} \wedge \mathtt{P'} \stackrel{s}{\Longrightarrow} \mathtt{R} \\ & \Rightarrow \exists \mathtt{P'} \in Spec : \mathtt{P} \stackrel{\alpha}{\rightarrow} \mathtt{P'} \wedge \mathtt{P'} \stackrel{s}{\Longrightarrow} \mathtt{R} \\ & \Rightarrow \mathtt{P} \stackrel{t}{\Longrightarrow} \mathtt{R} \\ & \Rightarrow t \in traces(\mathtt{P}) \subseteq traces(\mathtt{P}) \cup traces(\mathtt{Q}) \end{aligned}$$

     (b) $\nexists \mathtt{P'} : \mathtt{P} \stackrel{\alpha}{\rightarrow} \mathtt{P'}$

     $$\begin{aligned} & \Rightarrow \exists \mathtt{Q'} \in Spec : \mathtt{P\ []\ Q} \stackrel{\alpha}{\rightarrow} \mathtt{Q'} \wedge \mathtt{Q'} \stackrel{s}{\Longrightarrow} \mathtt{R} \\ & \Rightarrow \exists \mathtt{Q'} \in Spec : \mathtt{Q} \stackrel{\alpha}{\rightarrow} \mathtt{Q'} \wedge \mathtt{Q'} \stackrel{s}{\Longrightarrow} \mathtt{R} \\ & \Rightarrow \mathtt{Q} \stackrel{t}{\Longrightarrow} \mathtt{R} \\ & \Rightarrow t \in traces(\mathtt{Q}) \subseteq traces(\mathtt{P}) \cup traces(\mathtt{Q}) \end{aligned}$$

(c) sonst:
$(\exists \mathtt{Q}' : \mathtt{Q} \xrightarrow{\alpha} \mathtt{Q}') \wedge (\exists \mathtt{P}' : \mathtt{P} \xrightarrow{\alpha} \mathtt{P}') \wedge (\mathtt{P}' \ \mathtt{[]}\ \mathtt{Q}' \xRightarrow{s} \mathtt{R})$

$\Rightarrow (s \in \mathit{traces}(\mathtt{P}') \cup \mathit{traces}(\mathtt{Q}')) \wedge$

a) entweder $s \in \mathit{traces}(\mathtt{P}') \Rightarrow t \in \mathit{traces}(\mathtt{P})$

b) oder $s \in \mathit{traces}(\mathtt{Q}') \Rightarrow t \in \mathit{traces}(\mathtt{Q})$

$\Rightarrow t \in \mathit{traces}(\mathtt{P}) \cup \mathit{traces}(\mathtt{Q})$

- Sei $t \in \mathit{traces}(\mathtt{P}) \cup \mathit{traces}(\mathtt{Q})$. Zu zeigen: $t \in \mathit{traces}(\mathtt{P}\ \mathtt{[]}\ \mathtt{Q})$.
  Betrachte folgende Fälle:

  (a) $t \in \mathit{traces}(\mathtt{P})$

  $\Rightarrow \exists \mathtt{P}' : \mathtt{P} \xrightarrow{\alpha} \mathtt{P}' \wedge s \in \mathit{traces}(\mathtt{P}')$

  i. $\nexists \mathtt{Q}' : \mathtt{Q} \xrightarrow{\alpha} \mathtt{Q}'$

  $\Rightarrow \mathtt{P}\ \mathtt{[]}\ \mathtt{Q} \xrightarrow{\alpha} \mathtt{P}'$

  $\Rightarrow t \in \mathit{traces}(\mathtt{P}\ \mathtt{[]}\ \mathtt{Q})$

  ii. $\exists \mathtt{Q}' : \mathtt{Q} \xrightarrow{\alpha} \mathtt{Q}'$

  $\Rightarrow \mathtt{P}\ \mathtt{[]}\ \mathtt{Q} \xrightarrow{\alpha} \mathtt{P}'\ \mathtt{[]}\ \mathtt{Q}'$
  $\wedge s \in \mathit{traces}(\mathtt{P}') \cup \mathit{traces}(\mathtt{Q}')$

  $\Rightarrow \mathtt{P}\ \mathtt{[]}\ \mathtt{Q} \xrightarrow{\alpha} \mathtt{P}'\ \mathtt{[]}\ \mathtt{Q}'$
  $\wedge s \in \mathit{traces}(\mathtt{P}'\ \mathtt{[]}\ \mathtt{Q}')$

  $\Rightarrow \mathtt{P}\ \mathtt{[]}\ \mathtt{Q} \xrightarrow{\alpha} \mathtt{P}'\ \mathtt{[]}\ \mathtt{Q}'$
  $\wedge \exists \mathtt{R} : \mathtt{P}'\ \mathtt{[]}\ \mathtt{Q}' \xRightarrow{s} \mathtt{R}$

  $\Rightarrow t \in \mathit{traces}(\mathtt{P}\ \mathtt{[]}\ \mathtt{Q})$

  (b) $t \in \mathit{traces}(\mathtt{Q})$
  analog

□

Äquivalenzen: Für die Auswahl gelten folgende Äquivalenzen:

$$
\begin{aligned}
&\mathtt{P}\ \mathtt{[]}\ \mathtt{P} && =_T \mathtt{P}\\
&\mathtt{P1}\ \mathtt{[]}\ (\mathtt{P2}\ \mathtt{[]}\ \mathtt{P3}) && =_T (\mathtt{P1}\ \mathtt{[]}\ \mathtt{P2})\ \mathtt{[]}\ \mathtt{P3}\\
&\mathtt{P1}\ \mathtt{[]}\ \mathtt{P2} && =_T \mathtt{P2}\ \mathtt{[]}\ \mathtt{P1}\\
&\mathtt{P}\ \mathtt{[]}\ \mathtt{STOP} && =_T \mathtt{P}
\end{aligned}
$$

□

3.2.4.7 Parallel-Komposition

Die Parallel-Komposition `P [A||B] Q` besteht aus Prozess `P` (verarbeitet Ereignisse aus `A`) und `Q` (verarbeitet Ereignisse aus `B`). Die Prozesse `P` und `Q` synchronisieren auf $\mathtt{A} \cap \mathtt{B}$ und führen alle andere Ereignisse unabhängig aus.
Da `P` alle Ereignisse aus `A` verarbeitet, wird auch die Projektion auf `A` aller Ereignisse, die von der Parallel-Komposition verarbeitet werden, von `P` ausgeführt. Dies gilt analog für alle Ereignisse aus der Projektion auf `B`, die von `Q` verarbeitet werden. Die Menge der Traces von `P [A||B] Q` wird durch die Sequenzen von Ereignissen dargestellt, die sowohl mit `P` als auch mit `Q` konsistent sind. Es werden ausschließlich Ereignisse, die in `A` oder `B` enthalten sind und das Terminierungs-Ereignis ✓ verarbeitet.

$$\begin{aligned} \mathit{traces}(\mathtt{P\ [A||B]\ Q}) = \{s : \mathrm{seq}(\mathtt{A} \cup \mathtt{B} \cup \{\checkmark\}) \mid \\ s \upharpoonright (\mathtt{A} \cup \{\checkmark\}) \in \mathit{traces}(\mathtt{P}) \\ \wedge \\ s \upharpoonright (\mathtt{B} \cup \{\checkmark\}) \in \mathit{traces}(\mathtt{Q})\} \end{aligned}$$

3.2.4.8 Überwachter Aufruf

Die Trace des Prozess-Identifikators `X` werden durch den Prozess `P` definiert, durch welchen `X` deklariert ist. Enthält die Deklaration von `X` Rekursion, ist darauf zu achten, dass zwischen zwei Aufrufen mindestens ein Ereignis verarbeitet wird. Beispielsweise ist die Deklaration `X = X` nicht gestattet. Diese Gleichung würde alle Traces erlauben und ist somit für die Spezifikation von Java-Programmen wenig hilfreich. Die Trace-Semantik des überwachten Aufrufs lautet:

$$\mathit{traces}(\mathtt{X}) = \mathit{traces}(\mathtt{P})$$

wobei `X` durch `P` deklariert ist.

3.3 Spezifikation von Zeit

Jedes Jassda-Ereignis enthält neben den Informationen über das untersuchte Programm auch eine Menge von Zeitstempeln. Die Zeitstempel werden zum Zeitpunkt, an dem das Ereignis erzeugt wird, von eine Menge von Uhren abgelesen und bleiben während der Verarbeitung des Ereignisses konstant. Die Information Zeit kann somit wie alle anderen Attribute eines Ereignisses ausgewertet werden. Jassda stellt zwei Typen von Uhren bereit, um das zeitliche Verhalten eines Programmes zu spezifizieren: Echtzeit-Uhren und Laufzeit-Uhren.

3.3.1 Echtzeit

Die Echtzeit-Uhren messen die real vergangene Zeit. Sie können zum Beispiel zur Messung der Antwortzeiten externer Komponenten, wie Datenbanken, Benutzerinteraktionen, etc. eingesetzt werden.

3.3.2 Laufzeit

Laufzeit-Uhren messen nur die Zeit, in der das untersuchte Programm (Debuggee) tatsächlich arbeitet. Sobald Jassda ein Ereignis vom Debuggee empfängt, wird dieser angehalten, und die Laufzeit-Uhren werden gestoppt. Sobald alle von Jassda initiierten Operationen (Trace überprüfen, etc.) abgeschlossen sind, wird der Debuggee wieder angestoßen; gleiches gilt für die Laufzeit-Uhren.

3.3.3 Operatoren

Der Zugriff auf die Uhren geschieht durch spezielle Ereignismengen. Diese erlauben folgende Operationen auf den Uhren:

Init: Die Uhr wird mit dem Wert einer Konstanten oder einer anderen Uhr initialisiert werden.

Vergleich: Der Wert einer Uhr kann mit einer Konstanten oder dem Wert einer anderen Uhr verglichen werden.

Beispiel: Ein Roboter klebt zwei Werkstücke zusammen. Der Kleber benötigt mindestens 5 Sekunden, um abzubinden. Er muss die Werkstücke also mindestens 5 Sekunden zusammenpressen. Dies läßt sich in CSP_{jassda} wie folgt definieren:

$$\texttt{P()} = \texttt{robot.press.begin.timer}_{timer1=0}$$
$$\texttt{-> robot.press.end.timer}_{timer1>5s}$$
$$\texttt{-> P()}$$

□

3.4 Validierung von Trace- und Zeit-Zusicherungen

In den vorherigen Abschnitten wurde beschrieben, wie Abläufe in CSP_{jassda} spezifiziert werden. Nun wird hier erläutert, wie geprüft wird, ob sich eine in Java implementierte Applikation gemäß dieser Spezifikation verhält.

Um zu validieren, ob ein Java-Programm *Proc* die CSP_{jassda} -Spezifikation *Spec* erfüllt ist nachzuweisen, ob

$$traces(Prog) \subseteq traces(Spec)$$

gilt, also ob die Menge aller Traces des Programmes in der Menge aller Traces der Spezifikation enthalten ist. Jassda liefert nur solche Ereignisse, die auch im Alphabet der Spezifikation enthalten sind:

$$\alpha(Prog) \subseteq \alpha(Spec)$$

Jassda erzeugt zur Laufzeit eines untersuchten Programmes eine Sequenz von Ereignissen, welche eine Trace des Programmes darstellen. Mit Jassda kann also nur ein konkreter Ablauf eines Programmes validiert werden. Das heißt, es wird überprüft, ob

$$tr \in traces(Spec)$$

wobei $tr \in traces(Prog)$.

Mit anderen Worten: Die CSP_{jassda} -Spezifikation definiert eine Menge gültiger Traces. Jassda prüft, ob eine Folge von Ereignissen, die durch ein ablaufendes Programm erzeugt wird, ein nach CSP_{jassda} -Spezifikation gültige Trace ist ist.

Die Validierung einer Trace des untersuchten Programmes geschieht durch Abarbeiten (Interpretieren) der Spezifikation, welche durch einen CSP_{jassda} -Prozess gegeben ist. Für jedes Ereignis, welches Jassda bei der Beobachtung des untersuchten Programmes liefert, wird versucht, den CSP_{jassda} -Prozess auszuführen. Ist die Ausführung erfolgreich, repräsentiert der neu entstandene CSP_{jassda} -Prozess die Menge der zu diesem Zeitpunkt möglichen Traces gemäß der Trace-Semantik. Andernfalls wird ein Fehler angezeigt.

Bemerkung: Das Erzeugen von Ereignissen sowie deren Verarbeitung verursacht einen Overhead, welcher das zeitliche Verhalten des untersuchten Programmes verfälscht. Die Ausführungsgeschwindigkeit verringert sich. Dies ist bei der Spezifikation von zeitlichen Bedingungen zu berücksichtigen. Hält ein Programm während es mit Jassda untersucht wird, eine *obere Zeitschranke* ein, so hält es diese Schranke auch dann ein, wenn Jassda nicht prüft. Hält das untersuchte Programm hingegen eine *untere Zeitschranke* ein, lässt sich daraus nicht ableiten, dass das ungeprüfte Programm (welches schneller ausgeführt werden kann) diese Schranke auch einhält.

Kapitel 4

Implementierung

Dieses Kapitel beschreibt die Implementierung der *Java with Assertions Debugger Architecture*. Im Abschnitt 4.1 wird zunächst auf softwaretechnische Überlegungen zur Umsetzung von Jassda eingegangen. Anschließend beschreiben die Abschnitte 4.2 bis 4.5 die Architektur, deren Komponenten und Funktionsweise. Abschließend werden in Abschnitt 4.6 ausgewählte Implementierungsdetails vorgestellt.

4.1 Einleitung

Wie in Kapitel 3 beschrieben, müssen zur Prüfung von Trace- und Zeit-Zusicherungen zwei Aufgabenbereiche abgedeckt werden:

1. Die Trace eines Programmes muss ermittelt werden.

2. Die ermittelte Trace muss ausgewertet werden.

Die Lösung dieser Aufgaben wird nun diskutiert.

4.1.1 Ermittlung einer Trace

Eine Trace eines Java-Programmes ist eine Abfolge von Methodenaufrufen. Zur Ermittlung einer Trace ist es notwendig, Auskunft über den Aufruf und das Ende der Methoden zu erlangen. Zunächst wird der Ansatz des Tools Jass (siehe Kapitel 2.2.2.4) diskutiert. Anschließend wird die Funktionsweise von Jassda beschrieben.

4.1.1.1 Jass

Das Tool Jass, welches im Rahmen der Diplomarbeiten von Bartetzko (1999) und Plath (2000) entwickelt wurde, fügt an Methoden-Anfängen und Ende Java-Code in den Quelltext ein, welcher eine zentrale Stelle über den Programmablauf informiert. Diese prüft zur Laufzeit, ob die ermittelte Trace gültig ist und zeigt gegebenenfalls einen Fehler an. Wie bereits in Kapitel 2 und speziell im Abschnitt 2.2.2.4 beschrieben wurde, hat der in Jass gewählte Ansatz einige Einschränkungen:

- Der Quelltext des untersuchten Programmes muss verfügbar sein.
- Der Programmcode wird instrumentiert. Durch eventuelle Seiteneffekte des ergänzten Codes kann sich die Funktionsweise des modifizierten Programmes vom nicht modifizierten unterscheiden.
- Um Zusicherungen zu verändern, ist eine Neuübersetzung des Quelltextes notwendig.

Desweiteren sind die Trace-Zusicherungen von Jass nicht für den gesamten Sprachumfang von Java anwendbar:

- Mehrere Threads werden nicht unterstützt.
- Das Konzept der Exceptions von Java wird nicht berücksichtigt.

Die Umsetzung von Trace- und Zeit-Zusicherungen durch Erweiterung des bestehenden Tools Jass erscheint im Hinblick auf die grundsätzlichen Einschränkungen des Ansatzes nicht praktikabel.

4.1.1.2 Jassda

Die *Java with Assertions Debugger Architecture* nutzt das Java Debug Interface (JDI) (siehe Kapitel 3.1.3.3), um Auskunft über die Trace des untersuchten Programmes zu erhalten. Dieser Ansatz hat den Vorteil, dass auch Java Programme untersucht werden können, deren Quelltext nicht vorhanden ist. Weiterhin wird der geprüfte Code nicht modifiziert.[1] Fehler durch eingefügten Code werden somit ausgeschlossen. Die Zusicherungen lassen sich zur Laufzeit des untersuchten Programmes ein- und ausschalten. Das JDI unterstützt die Untersuchung von Programmen, die in einer anderen Virtual Machine oder sogar auf einem anderen Rechner ablaufen (auch Remote-Debugging genannt). Dies ist dann interessant, wenn Probleme mit der Software beim Kunden auftreten, die nur in dessen Umgebung auftreten. Mit Jassda könnte der Fehler aus der Ferne untersucht werden.

[1] Eine geringfügige Modifikation des Bytecodes ist für die Ermittlung des Rückgabewertes einer Methode notwendig (siehe Kapitel 4.6.1)

Darüber hinaus unterstützt das JDI auch die gleichzeitige Kontrolle mehrerer Anwendungen. Somit lassen sich verteilte Applikationen und deren Kommunikation untersuchen.
Jassda konfiguriert die Virtual Machine (VM), in der das untersuchte Programm ausgeführt wird. Sobald der VM eine Methode aufruft oder beendet wird dies durch ein Ereignis angezeigt. Gleiches gilt für das Auftreten einer Exception. Nach jedem Ereignis ist die VM angehalten, und das Ereignis von Jassda ausgewertet. Nach der Auswertung wird die VM wieder gestartet.

4.1.2 Prüfen einer Trace

Zur Prüfung des ermittelten Traces stellt Jassda einen Trace-Prüfer bereit. Jedes Ereignis, welches das JDI liefert, wird um einen Satz von Uhren erweitert und an den Trace-Prüfer geschickt. Dieser hat eine CSP_{jassda} -Spezifikation (siehe Kapitel 3.2) eingelesen und daraus eine interne Datenstruktur aufgebaut. Diese Datenstruktur stellt einen CSP_{jassda} -Prozess dar, welcher die Menge der erlaubten Traces beschreibt. Für jedes Ereignis wird geprüft, ob es von diesem Prozess akzeptiert werden kann.

4.2 Java with Assertions Debugger Architecture

Beim Entwurf der *Java with Assertions Debugger Architecture* wurde auf eine möglichst weitreichende Wiederverwendbarkeit geachtet. Die Architektur ist in mehrere Komponenten untergliedert, die über drei Schichten verteilt sind. Die Abbildung 4.1 stellt die Architektur grafisch dar. Die Schichten und Komponenten werden nun näher erläutert.

Die Architektur ist in folgende drei Schichten gegliedert:

Debuggee Diese Schicht nimmt die untersuchten Programme (auch Debuggees genannt) auf. Sie stellt den anderen Schichten eine einheitliche Schnittstelle für den Zugriff auf die Debuggees zur Verfügung. Die Debuggees werden mit Hilfe des Java Debug Interface (JDI) derart konfiguriert, dass sie durch Erzeugen von Ereignissen über den Programmablauf des Debuggees Auskunft geben.

Core In der Core-Schicht werden die Ereignisse, die von der Debuggee-Schicht geliefert werden, aufbereitet und zur weiteren Auswertung an die Module weitergeleitet. Darüber hinaus werden hier die Komponenten aus der Debuggee- und Modul-Schicht verwaltet.

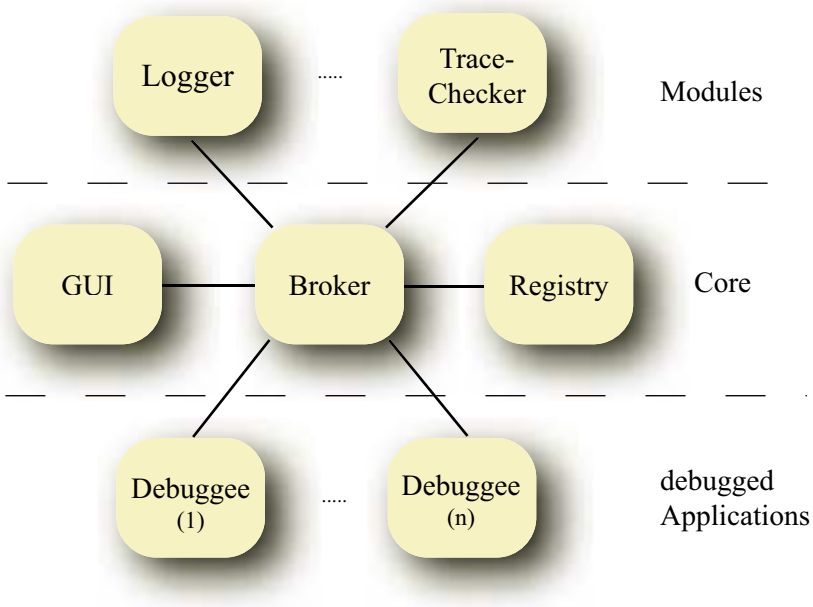

Abbildung 4.1: Java With Assertions Debugger Architecture

Module Die Architektur kann mehrere Module aufnehmen, welche die von der Core-Schicht gelieferten Ereignisse auswerten und/oder aufbereiten.

Bemerkung: Die Implementierung der *Java with Assertions Debugger Architecture* stellt mit der Debuggee- und Core-Schicht ein Framework bereit, welches als Basis für Applikationen zur Überprüfung von Java-Applikationen zur Laufzeit genutzt werden kann. Es kann die unterschiedlichsten Module aufnehmen. Im Rahmen dieser Arbeit ist ein Modul zur Ausgabe der Trace in eine Datei (Logger) sowie ein Modul zur Prüfung der Trace (Trace-Prüfer) implementiert.

Nachfolgend werden die Schichten und ihre Komponenten genauer beschrieben.

4.2.1 Debuggee-Schicht

Die Debuggee-Schicht kapselt die untersuchten Programme. Es wird von implementierungsnahen Details der Verbindung zu den untersuchten Programmen und deren Konfiguration abstrahiert. Die Verbindung mit den Virtual Machines der untersuchten Programme (Debuggees) kann durch Wahl verschiedener *Connectoren* (siehe 3.1.5.1) geschehen.

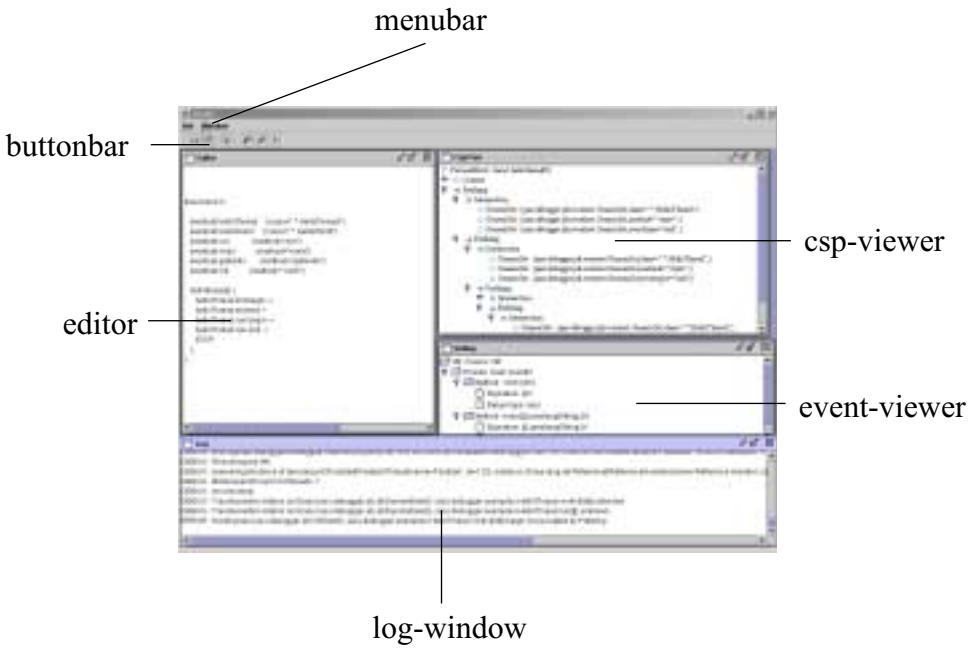

Abbildung 4.2: Grafische Benutzungsoberfläche von Jassda

4.2.2 Core-Schicht

Die Core-Schicht dient hauptsächlich der Konfiguration und Verwaltung der Module und Debuggees. Sie sammelt Ereignisse der einzelnen Debuggee-Komponenten aus der Debuggee-Schicht, filtert und verarbeitet sie und leitet sie an die Module weiter. Sie besteht aus einer grafischen Benutzungsoberfläche, einem Broker sowie einer Registrierungsdatenbank. Diese Komponenten werden nachfolgend näher beschrieben.

4.2.2.1 Grafische Benutzungsoberfläche

Die grafische Benutzungsoberfläche basiert auf dem Multiple Document Interface (MDI) der Java Swing API (siehe Robinson und Vorobiev 2000). Die Module und Debuggees werden grafisch durch ein oder mehrere Fenster innerhalb des MDI-Hauptfensters dargestellt.
Die Abbildung 4.2 zeigt die einzelnen Bereiche der Jassda Benutzungsoberfläche:

menubar Die Menü-Zeile erlaubt neben dem Zugriff auf allgemeine Datei-Operationen auch das Ein- und Ausblenden von Fenstern.

buttonbar Die Buttons erlauben den schnellen Zugriff auf auf den ”Laden” und ”Speichern”-Dialog. Mit dem ”Start”-Knopf können die Debuggees gestartet werden.

log-window Das Log-Fenster stellt die Ausgaben des Logging-Systems dar.

event-viewer Das aktuelle Ereignis wird im ”event-viewer” angezeigt.

Neben den Bereichen, welche die Benutzungsoberfläche des Jassda-Frameworks bietet, sind in Abbildung 4.2 auch die Fenster des Trace-Prüfer-Moduls (siehe Abschnitt 4.5) dargestellt:

editor Im Editor kann eine CSP_{jassda} -Spezifikation erstellt und verändert werden.

csp-viewer Der ”csp-viewer” zeigt die interne Datenstruktur der CSP_{jassda} -Spezifikation.

4.2.2.2 Broker

Der Broker stellt eine zentrale Komponente des Jassda-Frameworks dar. Seine Aufgabe besteht darin, die Module und Debuggees zu konfigurieren und zu verwalten. Außerdem sammelt er zur Laufzeit die Ereignisse der Debuggees, bereitet sie auf, filtert sie und erzeugt neue, welche an die Module weitergeleitet werden.

4.2.2.3 Registrierungsdatenbank

Die Registrierungsdatenbank (siehe Abbildung 4.3) speichert die Konfiguration des Jassda-Frameworks und stellt Datenstrukturen für die Verarbeitung von Ereignissen zur Laufzeit bereit. Folgende Komponenten realisieren die Registrierungsdatenbank.

Configuration Die `Configuration`-Komponente wird beim Start des Frameworks mit den Daten der Konfigurations-Datei initialisiert. Diese Daten können zur Laufzeit über die grafische Benutzungsoberfläche editiert werden. Im einzelnen hält die Configuration folgende Daten vor:

- Konfigurationsdaten für den Logging-Mechanismus. Dieser kann Informationen über den Programmablauf von Jassda protokollieren. Die Füller der protokollierten Informationen kann eingestellt werden.
- Konfigurationsdaten für die Debuggees. Hier wird definiert, wie Jassda mit welchen Programmen Kontakt aufnimmt.

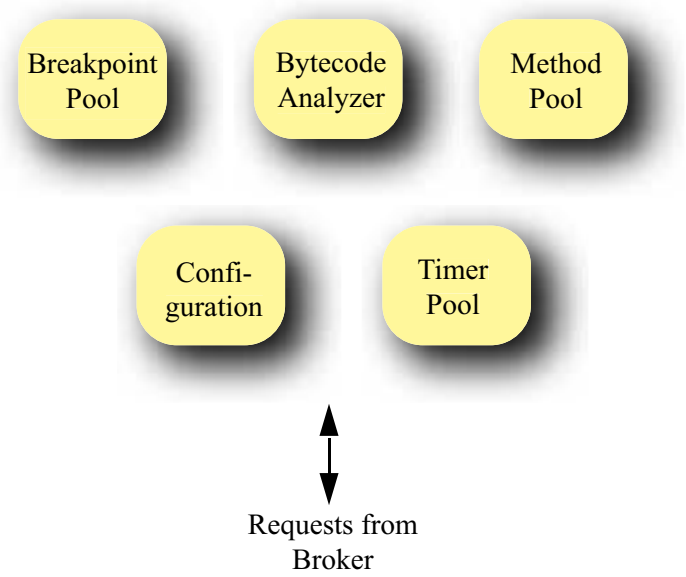

Abbildung 4.3: Registrierungsdatenbank

- Die einzelnen Module, die das Framework aufnehmen soll, werden aufgelistet.

Breakpoint Pool Um über Methoden-Anfänge und Enden informiert zu werden, werden im untersuchten Programm für alle Methoden, die ausgewertet werden sollen, am Anfang und am Ende Breakpoints gesetzt. Das JDI liefert daraufhin nach dem Aufruf und vor dem Beenden einer Methode ein `BreakpointEvent`. Dieses beinhaltet zwar genaue Information über den Breakpoint, welcher das Ereignis ausgelöst hat. Ob sich der Breakpoint am Anfang oder am Ende der Methode befunden hat, wird nicht mitgeteilt. Aus diesem Grund verwaltet der Breakpoint Pool eine Liste, die den Breakpoints einem Methoden-Anfang oder einem Methoden-Ende zuordnet.

Method Pool Nicht für alle Methoden in der Virtual Machine werden Breakpoints gesetzt. Dies würde den Programmablauf des Debuggees unnötig verzögern. Um nur diejenigen Ereignisse auszuwählen, die in der Module-Schicht ausgewertet werden können, wird für jede einzelne Methode jeder Klasse entschieden, ob Breakpoints gesetzt werden oder nicht. Für jede Methode jeder geladenen Klasse werden die Module gefragt, ob Ereignisse dieser Methoden verarbeitet werden sollen. Bei dieser Analyse werden alle neuen Methoden der untersuchten Klasse sowie alle geerbten Methoden betrachtet.

Angenommen zwei Klassen (`A` und `B`) erben von der Java Super-Klasse `java.lang.Object` und überschreiben nicht die `toString()` Methode. Dann würde bei der Untersuchung der Klassen `A` und `B` jeweils auch die Methode `toString()` der Super-Klasse `Object` analysiert werden. Der Method Pool verwaltet eine Liste bereits untersuchter Methoden und vermeidet somit mehrfache Analysen derselben Methoden.

Bytecode-Analyzer Jassda setzt Breakpoints an den Anfang und das Ende von Methoden, um über den Aufruf oder das Beenden benachrichtigt werden zu können. Der Bytecode-Analyzer hilft, die Positionen für die zu setzenden Breakpoints im Bytecode zu finden.

Clock Pool Die *Java with Assertions Debugger Architecture* sieht vor, dass auch das Laufzeitverhalten der Debuggees untersucht werden kann. Dazu verwaltet und manipuliert der Clock Pool eine Menge von Uhren. Diese können für Zeitmessungen zwischen einzelnen Ereignissen herangezogen werden. Es ist zwischen zwei Uhren-Type zu unterscheiden (siehe auch Abschnitt 3.3):

- Echtzeit-Uhren messen die real vergangene Zeit.
- Laufzeit-Uhren messen nur die Zeit, in der die untersuchte Applikation arbeitet. Die Zeit, die zur Verarbeitung von Ereignissen oder für andere Jassda-interne Operationen benötigt wird, wird nicht gemessen.

4.2.3 Module-Schicht

Das Jassda-Framework kann in der Modul-Schicht mehrere Module aufnehmen. Jedes einzelne Modul wird vom Broker durch Ereignisse vom aktuellen Ablauf der untersuchten Programme unterrichtet. Im Rahmen dieser Arbeit sind zwei Module entstanden: ein Modul, welches die Ereignisse der Reihe nach in eine Datei schreibt (*Logger*) und eines, welches die Trace eines Programmes zur Laufzeit prüft (*Trace-Prüfer*). An dieser Stelle wird nur ein kurzer Überblick gegeben. Nähere Details sind in den Abschnitten 4.4 und 4.5 zu finden.

4.2.3.1 Logger

Das Logger-Modul zeigt eine einfache Anwendung des Jassda-Frameworks. Neben der Auswahl der zu protokollierenden Ereignisse kann auch das Ausgabe-Format des Logger-Moduls konfiguriert werden.

4.2.3.2 Trace-Prüfer

Das Trace-Prüfer-Modul prüft zur Laufzeit den Ablauf (*Trace*) eines oder mehrerer untersuchter Programme. Gültige Traces werden in der in Abschnitt 3.2 eingeführten Spezifikationssprache CSP_{jassda} definiert.

4.3 Funktionsweise des Frameworks

Nachdem die Komponenten der Architektur eingeführt wurden, wird in folgendem Abschnitt auf ihre Funktion eingegangen. Ein typischer Ablauf beim Einsatz von Jassda durchläuft zwei Phasen:

1. Die Initialisierungs-Phase beschreibt den Zeitraum zwischen dem Start von Jassda und dem Zeitpunkt an dem das erste Ereignis eines Debuggees an ein Modul geschickt wird.

2. Die Laufzeitphase beschreibt den Zeitraum in welchem der Debugge arbeitet.

4.3.1 Initialisierungsphase

Beim Starten von Jassda wird das Hauptfenster der grafischen Benutzungsoberfläche geöffnet und die Hauptkonfigurationsdatei eingelesen. Diese XML-Konfigurationsdatei enthält folgende Bereiche:

Logging Das Logging-System des Frameworks wird konfiguriert (Verzeichnis, Format der Logging-Ausgabe. etc.)

Module Name, Modulklasse sowie Konfigurationsdatei jedes Moduls werden angegeben.

Debuggee Der *Connector* und dessen Konfiguration für den Zugriff auf das zu untersuchende Programm werden konfiguriert.

Beispiel: Die XML-Konfiguration 4.3.1 zeigt eine Konfiguration des Frameworks. Der erste Abschnitt konfiguriert das Logging-System. Log-Dateien werden in das Verzeichnis geschrieben in welchem Jassda gestartet wurde. Der Log-Level ist auf *debug* gestellt, um möglichst viele Daten während des Programmablaufs zu protokollieren. Das `pattern`-Attribut beschreibt das Format in dem die Log-Informationen ausgegeben werden. In diesem Fall wird jede Log-Ausgabe mit einem Zeilenumbruch begonnen (`%n`), anschließend folgt die

Konfiguration 4.3.1 Beispiel-Konfiguration des Frameworks

```
<?xml version="1.0" encoding="UTF-8"?>
<debugger>
  <log dir="." priority="debug" pattern="%n %m ">
     <event>
        <template>%class%.%method%.%eventtype%</template>
     </event>
  </log>

  <module name="trace-checker"
          class="jass.debugger.module.trace.gui.TraceAssertionGui"
          configfile="test/config/trace-checker.xml"/>

  <debuggee name="Server localhost">
     <connector name="SocketAttach">
        <arg key="host" value="localhost"/>
        <arg key="port" value="8000"/>
     </connector>
  </debuggee>
</debugger>
```

eigentliche Nachricht (`%m`). Das Format, in welchem Ereignisse formatiert werden, wird durch die angegebene Formatvorlage beschrieben.[2] Der nächste Abschnitt listet die Module auf. Hier wird der Trace-Prüfer mit der Hauptklasse `jass.debugger.module.trace.gui.TraceAssertionGui` und der Konfigurationsdatei `test/config/trace-checker.xml` konfiguriert. Der letzte Bereich konfiguriert einen Debuggee. Die Verbindung mit der Virtual Machine des untersuchten Programmes wird mit dem *SocketAttach* Connector (siehe Kapitel 3.1.5.1) aufgebaut. Das Programm muss auf demselben Rechner `localhost` im Debug-Modus gestartet sein und auf eine Verbindung mit einem Debugger auf dem Port 8000 warten. □

Die folgenden Abschnitte beschreiben den Initialisierungsprozess der Module und Debuggees.

4.3.1.1 Initialisierung der Module

Für jedes in der Hauptkonfigurationsdatei spezifizierte Modul wird die angegebene Modulklasse geladen und unter Angabe der Modul-spezifischen

[2] Für weitere Informationen über die Definition des Ausgabeformats wird auf die Quelltext-Dokumentation von Jassda verwiesen

Konfigurationsdatei initialisiert. Die Modulklasse muss das Java-Interface `jass.debugger.jdi.EventListener` implementieren, um als gültiges Modul akzeptiert zu werden. Das Format der Konfigurationsdatei ist beliebig.

4.3.1.2 Initialisierung der Debuggees

In der Initialisierungsphase der Debuggee-Komponenten wird eine Verbindung zum untersuchten Programm (Debuggee) hergestellt und dessen Virtual Machine derart konfiguriert, dass sie für Methode-Anfänge und Enden Ereignisse erzeugt.

Aufbau der Verbindung zum Debuggee Die Hauptkonfigurationsdatei enthält für jede Debuggee-Komponente Informationen über die Verbindungsart (Angabe des *Connectors*) (siehe 3.1.5.1) sowie deren Konfiguration. Die Verbindungen werden aufgebaut und die Virtual Machines (VMs) der Debuggees angehalten. Jede VM wird derart konfiguriert, dass sie Auskunft über den Programmablauf gibt.

Bytecode-Analyse und Konfiguration von Ereignissen In der angehaltenen Java Virtual Machine sind bereits einige Java Klassen geladen. Um den Ablauf des untersuchten Programmes nicht unnötig zu aufzuhalten, werden nur für diejenigen Aktionen der Debuggee Virtual Machine Ereignisse erzeugt, welche für die Module von Interesse sind.
Die Auswahl geschieht durch folgenden Algorithmus:

1. Lese die Namen alle geladenen Klassen
2. Lese die Methoden (auch die geerbten) jeder Klasse
3. Erzeuge für jede Methode ein *Dummy-Ereignis*, welches den Methoden-Anfang beziehungsweise dessen Ende anzeigt. (Diese *Dummy-Ereignisse* haben dieselbe Struktur wie die zur Laufzeit erzeugten Ereignisse).
4. Jedes Modul wird gefragt, ob es an der Verarbeitung des Ereignisses interessiert ist.
5. Falls ein Modul an einem Ereignis interessiert ist, wird der Bytecode der dazugehörigen Methode vom Bytecode-Analyzer analysiert und mit Breakpoints versehen. Im Breakpoint-Pool wird gespeichert, ob der jeweilige Breakpoint einen Methoden-Anfang oder Ende repräsentiert. Der Method-Pool speichert die Java-Methoden, welche bereits untersucht wurden.

Neben den Breakpoint Ereignissen zur Anzeige eines Methoden-Anfangs und Endes werden weitere Ereignisse definiert. Diese zeigen an, dass eine neue Klasse geladen wird, eine Exception auftritt oder das Programm beendet wurde.

4.3.2 Laufzeitphase

Zur Laufzeit zeigen die Debuggees durch Ereignisse an, dass eine Methode aufgerufen oder beendet wurde oder dass eine Exception aufgetreten ist. Sobald ein Ereignis erzeugt wird, hält der Debuggee an. Der *Broker* reichert die Ereignisse um weitere Informationen an und leitet sie weiter an die Module. Der Debuggee wird erst dann wieder gestartet, wenn alle Module das Ereignis verarbeitet haben.

Das Programmende des Debuggees wird durch Ereignisse des Typs `VMDeathEvent` (Debuggee Virtual Machine wurde beendet) sowie `VMDisconnectedEvent` (Verbindung zum Debuggee wurde getrennt) angezeigt. Diese Ereignisse werden vom Trace-Prüfer als ✓-Ereignis (siehe 3.2) aufgefasst.

Die Java Virtual Machine kann zur Laufzeit dynamisch Klassen nachladen. Dies wird durch das `ClassPrepareEvent` angezeigt. Für jede nachgeladene Klasse werden wie oben beschrieben Breakpoints an den Anfang und an das Ende der Methode gesetzt.

4.4 Modul: Logger

Eine einfache Anwendung des Jassda-Frameworks stellt das Logger-Modul dar. Es besteht aus zwei Java-Klassen: `TraceLogger` und `TraceLoggerConfiguration`. Die Klasse `TraceLoggerConfiguration` lädt die XML-Konfigurationsdatei und bereitet die Informationen auf. Der `TraceLogger` nutzt die aufbereiteten Informationen, um die zu protokollierenden Ereignisse auszuwählen und das Ausgabe-Format der Ereignisse zu definieren.

Die XML-Konfigurationsdatei des Logger-Moduls ist in vier Bereiche gegliedert:

1. Im *file*-Bereich wird die Ausgabedatei definiert

2. Wie ein Ereignis in die Ausgabedatei geschrieben werden soll, ist mittels eines Templates im *event*-Bereich definiert. Das Template darf beliebige Zeichen enthalten. Ausdrücke der Form `%returnvalue%` oder `%method%` sind Platzhalter, welche bei der Ausgabe durch konkrete Daten der Ereignisse ersetzt werden. Für eine Liste der Platzhalter wird auf die Quelltext-Dokumentation des Logging-Pakets von Jassda verwiesen.

3. Im *includes*-Bereich werden alle Ereignisse definiert, die protokolliert werden sollen. Die Auswahl wird durch Java-Klassen realisiert, die durch das

Konfiguration 4.4.1 Beispiel-Konfiguration des Logger-Moduls

```
<?xml version="1.0" encoding="UTF-8"?>
<logger>
  <file name="logger.out"/>
  <event>
    <template>
    [%thread%] %class%.%method%(%signature%)(%arguments%).%eventtype%
        = %returnvalue%
    </template>
  </event>
  <include>
    <eventset class="jass.debugger.jdi.eventset.MethodSet"
              field="class"
              argument="jass.debugger.examples.*"/>
    <eventset class="jass.debugger.jdi.eventset.MethodSet"
              field="class"
              argument="jass.examples.*"/>
  </include>
  <exclude>
    <eventset class="jass.debugger.jdi.eventset.MethodSet"
              field="class"
              argument="sun.*"/>
    <eventset class="jass.debugger.jdi.eventset.MethodSet"
              field="class"
              argument="jass.examples.servlet.*"/>
  </exclude>
</logger>
```

class-Argument spezifiziert werden. Diese Klassen werden näher durch weitere Argumente der *eventset*-Einträge konfiguriert.

4. Der *excludes*-Bereich definiert die Ereignisse, die nicht protokolliert werden.

Konfiguration 4.4.1 zeigt eine Beispielkonfiguration des Logger-Moduls.

4.5 Modul: Trace-Prüfer

Das Trace-Prüfer Modul überprüft zur Laufzeit eines Programmes die Trace- und Zeit-Zusicherungen. Die erlaubten Traces werden in der Spezifikationssprache CSP_{jassda} definiert. Zunächst wird kurz die Architektur des Trace-Prüfer-Moduls

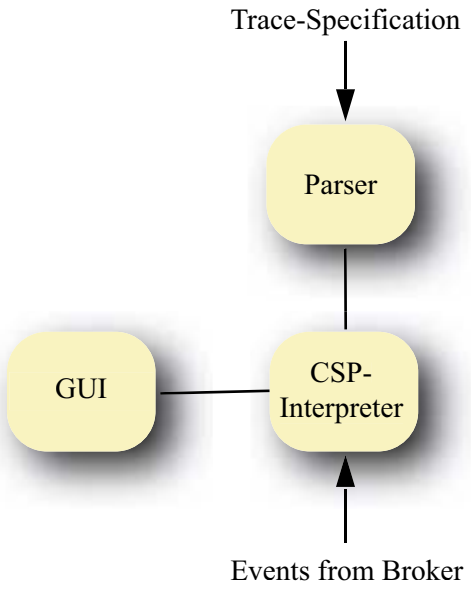

Abbildung 4.4: Trace-Prüfer

vorgestellt. Anschließend wird in Abschnitt 4.5.2 die Struktur der Spezifikationsdatei und der Parse-Vorgang beschieben. Der Abschnitt 4.5.3 geht auf die interne Repräsentation und Verarbeitung der Spezifikation ein. Die Konzepte der erweiterbaren Ereignismengen und Variablen folgt in den Abschnitten 4.5.4 und 4.5.5. Die Beschreibung der Implementierung der CSP_{jassda} -Operatoren folgt in Abschnitt 4.5.6. Im letzten Abschnitt wird schließlich anhand eines Beispiels erläutern, wie der Trace-Prüfer die Trace- und Zeit-Zusicherungen zur Laufzeit prüft.

4.5.1 Übersicht

Abbildung 4.4 skizziert die funktionalen Komponenten des Trace-Prüfers:

Parser Der Parser wandelt die CSP_{jassda} Spezifikation in eine interne Repräsentation um. Diese stellt einen CSP_{jassda} -Prozess dar und beschreibt die Menge der erlaubten Traces.

Interpreter Der Interpreter empfängt die vom Broker gesendeten Ereignisse und wandelt den CSP_{jassda} -Prozess entsprechend der operationellen Semantik von CSP_{jassda} um.

GUI Die grafische Benutzungsoberfläche stellt zwei Komponenten bereit:

1. In einem *Editor* kann die CSP_{jassda} -Spezifikation editiert werden.

2. Ein weiteres Fenster zeigt den aktuellen internen CSP_{jassda} -Prozess an und gibt Auskunft darüber, ob das letzte Ereignis verarbeitet werden konnte oder nicht.

4.5.2 Spezifikationssprache

Die in Kapitel 3.2 formal eingeführten Sprachkonstrukte von CSP_{jassda} werden hier leicht erweitert. Diese Erweiterungen sind ausschließlich syntaktischer Natur und sollen dem Schreiber einer CSP_{jassda} -Spezifikation die Übersicht erleichtern und Schreibarbeit abnehmen. Der Aufbau einer CSP_{jassda} -Spezifikationsdatei wird anhand eines Beispiels erläutert.

4.5.2.1 Beispiel

Die Spezifikation 4.5.1 auf Seite 77 zeigt eine CSP_{jassda} -Spezifikation für ein fiktives Applet, welches ein. Sie definiert zwei Traces:

lifecycle Die erste Trace spezifiziert den Lebenszyklus des Applets. Das Applet muss durch erfolgreiches Abarbeiten der `init()`-Methode initialisiert werden. Danach folgen alternierend Aufrufe der `start()` und `stop()`-Methoden. Die `destroy()`-Methode darf nur nach erfolgreichem Abschluss von `stop()` gestartet werden.

game Das Spiel ist für zwei Spieler ausgelegt und zeigt den jeweiligen Spielstand in getrennten Bildschirmbereichen (*leftscreen* und *rightscreen*) an. Der Spielablauf ist für beide Bereiche gleich: Als Erstes wird eine Eingabe vom Spieler erfragt. Anschließend wird der Bereich mit der `paint()`-Methode, welche `paintBalls()` und `paintLines()` nutzt, neu gezeichnet. Dies wird so lange wiederholt, bis das Applet terminiert.

4.5.2.2 Struktur der Spezifikationsdatei

Die Struktur der Spezifikationsdatei wird durch folgende BNF-Syntax beschrieben:

$$
\begin{array}{lll}
\textit{Specification} & ::= & (\textit{EventSetDeclaration})* \\
 & & [\textit{AlphabetDeclaration}] \\
 & & (\textit{TraceDeclaration})+ < \textit{EOF} > \\
\textit{TraceDeclaration} & ::= & ''\textit{trace}'' \ \textit{Label TraceBody} \\
\textit{TraceBody} & ::= & ''\{''(\textit{TraceBodyDeclaration})+''\}'' \\
\textit{TraceBodyDeclaration} & ::= & (\textit{EventSetDeclaration})* \\
 & & [\textit{AlphabetDeclaration}] \\
 & & (\textit{ProcessDeclaration})+ \\
\textit{EventSetDeclaration} & ::= & ''\textit{eventset}'' \ \textit{Label EventSet} \\
\textit{AlphabetDeclaration} & ::= & ''\textit{alphabet}'' \ [''+''] \ \textit{EventSet} \\
\textit{ProcessDeclaration} & ::= & \textit{Label FormalParameters ProcessBody} \\
\textit{ProcessBody} & ::= & ''\{'' \ \textit{ProcessBodyDeclaration} \ ''\}'' \\
\textit{ProcessBodyDeclaration} & ::= & (\textit{EventSetDeclaration})* \\
 & & [\textit{AlphabetDeclaration}] \\
 & & \textit{Process}
\end{array}
$$

Dabei bezeichnet *Label* den Namen einer Trace, einer Ereignismenge oder eines Prozesses. Die Ausdrücke *EventSetExpression* und *ProcessExpression* werden in den Kapiteln 4.5.4 und 4.5.6 beschrieben.

In der CSP_{jassda} -Spezifikationsdatei können mehrere Traces definiert werden. Eine Trace wird durch das Schlüsselwort `trace` und einem Bezeichner eingeleitet. Der Rumpf der Trace-Definition ist in Teilbereiche untergliedert:

- Referenzen für häufig genutzte *Ereignismengen* werden durch das Schlüsselwort `eventset`, einem Bezeichner sowie der referenzierten Ereignis-Menge angegeben.
- Optional kann das *Alphabet* der Trace definiert werden. Das Schlüsselwort `alphabet` wird gewählt, um das Alphabet zu *setzen*. Mit `alphabet+` wird das (implizit aus den definierten Prozessen) berechnete Alphabet *erweitert*.
- Ein oder mehrere Prozesse werden im Trace-Rumpf definiert. Der Prozess-Rumpf nimmt Prozess-Ausdrücke, wie sie in Kapitel 3.2 vorgestellt wurden, auf.

Intern wird ein Trace-Abschnitt als ein CSP_{jassda} -Prozess ausgewertet und beginnt mit dem `MAIN()`-Prozess. Die anderen Prozesse werden vom `MAIN()`-Prozess aufgerufen. Falls das Alphabet nicht explizit durch das `alphabet`-Schlüsselwort gesetzt wird, ergibt es sich aus der Vereinigung der Alphabete der in dem Trace-Abschnitt definierten Prozesse.

Spezifikation 4.5.1 CSP_{jassda} -Spezifikation eines Spiel-Applets

```
trace lifecycle {
    alphabet {instanceof="java.applet.Applet"}

    eventset start   {method="start"}
    eventset stop    {method="stop"}
    eventset destroy {method="destroy"}

    MAIN() {
       {method="init"}.begin
       -> {method="init"}.end
       -> RUN()
    }
    RUN() {
       start.begin -> start.end
       -> stop.begin -> stop.end
       -> (RUN() [] STOP())
    }
    STOP() {
       destroy.begin -> destroy.end
       -> TERM
    }
}

trace game {
    eventset leftscreen  {class="PanelLeft"}
    eventset rightscreen {class="PanelRight"}
    eventset paint       {method="paint"}
    eventset paintBalls  {method="paintBalls"}
    eventset paintLines  {method="paintLines"}

    MAIN() {
       init.start -> init.end
       -> PLAY()

    PLAY() {
       PLAY(leftscreen) || PLAY(rightscreen}
    }
    PLAY{screen} {
       screen.input.begin -> screen.input.end
       -> screen.paint.begin
       -> screen.paintBalls.begin -> screen.paintBalls.end
       -> screen.paintLines.begin -> screen.paintLines.end
       -> screen.paint.end
       -> (PLAY(screen) [] TERM)
    }
}
```

4.5.2.3 Parsen der Spezifikation

Der CSP_{jassda} -Parser des Trace-Prüfers wird automatisch mit Hilfe des JavaCC-Tools von *WebGain, Inc.* aus der CSP_{jassda} -Grammatik generiert. Kleine Änderungen und Anpassungen der Grammatik können somit mit geringem Implementierungsaufwand durchgeführt werden.

Syntax-Analyse JavaCC erzeugt zu einer gegebenen Grammatik einen Top-Down-Parser. Dieser generiert für eine CSP_{jassda} -Spezifikation im Hauptspeicher einen Syntaxbaum. Syntaktische Fehler in der CSP_{jassda} -Spezifikation werden angezeigt. Die Knoten des Syntaxbaumes werden durch Klassen des Typs `jass.debugger.modules.trace.parser.Node` implementiert. Jeder Knoten verfügt über Methoden, die Auskunft über den Vater und die Söhne geben. Weiterhin wird die Methode `jjtAccept` implementiert, welche die Anwendung des Besucher-Entwurfsmusters erlaubt.

Semantische-Analyse Der während der syntaktischen Analyse aufgebaute Syntaxbaum wird analysiert und in eine interpretierbare interne Datenstruktur überführt. Dies geschieht mit Hilfe einer Besucher-Klasse (Visitor), die für jeden Knotentyp eine angepasste Methode enthält. Beginnend mit dem Wurzel-Knoten wird der Baum durchlaufen. An jedem Knoten wird dessen Methode `jjtAccept` aufgerufen, welche die (für den Knoten zuständige) Methode des Visitors aufruft.
Der `jass.debugger.modules.trace.visitor.CspTreeVisitor` durchläuft den Syntaxbaum und erzeugt daraus eine *interne Datenstruktur* des in der Spezifikationsdatei definierten CSP_{jassda} -Prozesses. In einer *Symboltabelle* werden die Traces, Prozesse, Variablennamen und Ereignismengen abgelegt. Auf diese Weise kann leicht festgestellt werden, ob einzelne Einträge richtig genutzt werden. Beispielsweise müssen Ereignismengen, welche innerhalb eines Prozesses referenziert werden, vorher definiert worden sein.

4.5.3 Interpreter

Während der semantischen Analyse wird eine interne Repräsentation der Spezifikation aufgebaut. Diese Datenstruktur besteht aus einem CSP_{jassda} -Prozess und einer Symboltabelle, welche die in der Spezifikation deklarierten Ereignismengen und Prozesse aufnimmt. Jede Ereignismenge wird von einer Klasse des Typs `jass.csp.EventSet` dargestellt. Jeder Prozess-Operator wird durch eine Klasse des Typs `jass.csp.CSProcess` repräsentiert. Der interne CSP_{jassda} -Prozess beschreibt die Menge der erlaubten Traces. Der Interpreter verarbeitet ihn entsprechend der operationellen Semantik von CSP_{jassda} (siehe Kapitel 3.2.3). Im CSP_{jassda} -Prozess referenzierte Prozesse und Ereignismengen werden aus der

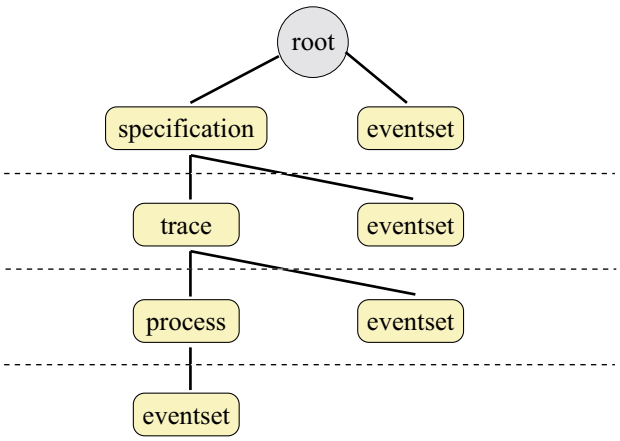

Abbildung 4.5: Hierarchische Struktur der Namensräume

Symboltabelle geladen.

Nachfolgend wird dargestellt, unter welchen Namen Prozesse und Ereignismengen in der Tabelle abgelegt sind.

4.5.3.1 Namensgebung: Prozesse

Die Spezifikation 4.5.1 deklariert sowohl in der Trace `lifecycle` als auch in der Trace `game` jeweils einen Prozess mit dem Namen `MAIN`. Um beide unterscheiden zu können wird ihr Name intern um einen Präfix erweitert, welcher sich aus der in Abbildung 4.5 dargestellten Hierarchie ableitet.
Außerdem deklariert die Trace `game` die Prozesse `PLAY()` und `PLAY(screen)`, welche sich in der Anzahl der übergebenen Parameter unterscheiden. Die Parameter eines CSP_{jassda} -Prozesses haben immer den Typ `jass.csp.EventSet`. Zur Unterscheidung reicht es somit aus, die Prozesse intern mit der Anzahl der Parameter zu versehen.

Beispiel: Wird die Spezifikation 4.5.1 beispielsweise aus einer Datei mit dem Name `applet` geladen, dann erhält der `MAIN`-Prozess der Trace `game` folgenden Namen:

```
applet|game|MAIN(0)
```

□

Gültigkeitsbereich	Beschreibung
Global	Die Ereignismengen gelten für jede Spezifikation. Die vordefinierten Ereignismengen `begin`, `end`, `any` und `exception` (siehe Kapitel 3.2.2.2) sind *global* gültig.
Spezifikation	Aus jedem Prozess jedes `trace`-Abschnitts der *Spezifikation* kann auf die Ereignismenge zugegriffen werden. Derartige Mengen werden am Anfang der Spezifikation vor dem ersten `trace`-Abschnitt definiert.
Trace	Werden die Ereignismengen innerhalb eines `trace`-Abschnitts definiert, sind sie nur für Prozesse innerhalb dieser Trace gültig.
Prozess	Ereignismengen, die am Anfang einer Prozess-Deklaration definiert werden, sind nur für diesen Prozess gültig.

Tabelle 4.1: Gültigkeitsbereiche von Ereignismengen

4.5.3.2 Namensgebung: Ereignismengen

Prozesse werden innerhalb eines `trace`-Abschnitts definiert und sind nur dort gültig. Es können keine Prozesse aus einer anderen Trace aufgerufen werden. Ereignismengen können hingegen für unterschiedliche Gültigkeitsbereiche definiert werden (siehe Abbildung 4.5 und Tabelle 4.1):

Die unterschiedlichen Gültigkeitsbereiche der Ereignismengen spiegeln sich in der Wahl der internen Namensgebung wieder.

Beispiel: Die vordefinierte *globale* Ereignismenge `begin` wird intern unter dem Namen `begin` verwaltet. Die Ereignismenge `start` welche innerhalb der Trace `lifecycle` in Spezifikation 4.5.1 definiert wird, wird intern unter dem Namen `applet|lifecycle|start` geführt. □

4.5.3.3 Suche von Ereignismengen in der Symboltabelle

Die Suche nach Ereignismengen in der Symboltabelle wird anhand eines Beispiels erläutert.

Beispiel: Im Prozess `applet|game|PLAYER(1)` wird die globale Ereignismenge `begin` referenziert. `Jassda` sucht zunächst nach einer Ereignismenge, die im Prozess `applet|game|PLAYER(1)` definiert ist und anschließend in den höheren Ebenen:

1. `applet|game|PLAYER(1)|begin`
2. `applet|game|begin`
3. `applet|begin`
4. `begin`

□

Prozess-Blöcke: Prozess-Blöcke beschreiben die Gültigkeitsbereiche von Prozess-Argumenten, Variablen und Alphabeten. Folgende Prozess-Blöcke werden von Jassda definiert:

1. Jede *Prozess-Deklaration* aus der Spezifikation ist ein Prozess-Block.
2. Die Sub-Prozesse des Auswahl-, Parallel- und Sequenz-Operators sind Prozess-Blöcke.

Beispiel: Ein Beispiel für einen Prozess-Block einer Prozess-Deklaration:

```
PLAY{screen} {
   screen.input.begin -> screen.input.end
   -> screen.paint.begin
   -> screen.paintBalls.begin -> screen.paintBalls.end
   -> screen.paintLines.begin -> screen.paintLines.end
   -> screen.paint.end
   -> (PLAY(screen) [] TERM)
}
```

Die Prozess-Deklaration `PLAY(screen)` stellt einen Prozess-Block dar. Das Prozess-Argument `screen` gilt für den gesamten Prozess.

Ein Beispiel für einen Prozess-Block eines Sub-Prozesses:

```
START(method) {
   leftscreen.method.begin
   -> leftscreen.method.end
   -> ANY[leftscreen]
   ||
   rightscreen.method.begin
   -> rightscreen.method.end
   -> ANY[rightscreen]
}
```

Der Prozess `START()` beschreibt, dass sowohl im linken als auch im rechten Bildschirmbereich als erstes die Methode `method` aufgerufen werden muss. Die Sub-Prozesse für den linken und rechten Bildschirmbereich stellen Prozess-Blöcke dar. Die Deklaration von `START()` ist ebenfalls ein Prozess-Block. □

4.5.4 Ereignismengen

Jassda akzeptiert komplexe Ausdrücke für die Spezifikation von Ereignismengen. Die folgende Syntax beschreibt den Aufbau der Ereignismengen:

EventSet ::= *EventSetInput*
| *EventSetUnion*
| *EventSetIntersection*
| *EventSetDefinition*
| *EventSetVariable*
| *EventsetReference*

EventSetUnion ::= *EventSet* (
(″,″ | ″+″) *EventSet*
)∗

EventSetIntersection ::= *EventSet* (
(″.″ | ″!″) *EventSet*
)∗

EventSetDefinition ::= ″{″
Parameter = *String*
(″,″ *Parameter* = *String*)∗
″}″

Dabei beschreibt *EventSetInput* die Wertzuweisung einer Variablen (siehe Kapitel 4.5.5), *EventSetUnion* die Vereinigung, *EventSetIntersection* den Schnitt, *EventSetDefinition* die Definition einer Ereignismenge (siehe Abschnitt 4.5.4.1), *EventSetReference* einen Verweis auf eine vorher definierte Ereignismenge und *EventSetVariable* einen Variablennamen.

4.5.4.1 Definition von Ereignismengen

Im Kapitel 3.2 wurden Ereignismengen durch eine Abbildung f definiert, die angibt, ob ein konkretes Ereignis in der Menge enthalten ist oder nicht. Um Ereignismengen möglichst flexibel definieren zu können, wird die Abbildung f nicht durch einen festen Mechanismus sondern durch konfigurierbare Java-Klassen implementiert. Für häufig genutzte Ereignismengen stellt Jassda

Standard-Implementierungen bereit. Problemlos können für spezielle Bedürfnisse weitere Klassen für Ereignismengen implementiert werden. In der CSP_{jassda} - Spezifikation wird eine Ereignismenge durch den Namen der implementierenden Klasse, sowie weiteren konfigurierenden Parametern deklariert. Der Parameter `handler` ist für die Definition der implementierenden Klasse reserviert. Um dem Ersteller der Spezifikation unnötige Schreibarbeit abzunehmen, kann in der Konfigurationsdatei des Trace-Prüfers eine Default-Klasse eingetragen werden, die gewählt wird falls kein `handler`-Parameter angegeben wird. Weiterhin können für lange Klassennamen Aliase definiert werden. Lange Klassennamen können somit durch kurze Bezeichner referenziert werden. Alle weiteren Parameter dienen zur Konfiguration der implementierenden Klasse.

Beispiel: Die Menge aller Ereignisse, die von der Methode mit dem Namen `toString` beliebiger Klassen erzeugt werden, kann durch

```
{handler="jass.debugger.eventset.GenericSet",
 method="toString"}
```

modelliert werden. Falls in der Konfigurationsdatei als Default-Ereignismenge die Klasse `jass.debugger.eventset.GenericSet` eingetragen ist, lässt sich dies auch durch

```
{method="toString"}
```

abkürzen. □

4.5.4.2 Standard-Ereignismengen

Jassda stellt zwei Implementierungen von Ereignismengen bereit:

Die Ereignismenge GenericSet: Die Klasse `GenericSet` ist zur Definition von einfachen Ereignismengen geeignet. Sie unterstützt unter anderen die Parameter `class` (Name der Klasse), `method` (Name der Methode) und `instanceof` (implementiertes Interface oder Super-Klasse). Für eine Liste der unterstützten Parameter wird auf die Quelltext-Dokumentation der Klasse `jass.debugger.eventset.GenericSet` verwiesen,

Beispiel: Alle Ereignisse aus dem Paket `jass.debugger` werden durch

```
{class="jass.debugger.*"}
```

beschrieben. Die Menge aller Ereignisse, die von der Methode `paint` einer von `java.awt.Component` abgeleiteten Klasse erzeugt werden können, wird durch

```
{method="paint",
 instanceof="java.awt.Component"}
```

definiert. □

Die Ereignismenge TimeSet: Die Ereignismenge `TimeSet` implementiert die Interaktion des Trace-Prüfers mit dem Uhren-Pool des Jassda-Frameworks. Zusicherungen, die Aussagen über das zeitliche Verhalten eines Programmes machen, lassen sich hiermit realisieren. Beispielsweise kann mit ihr ausgedrückt werden, dass zwischen dem Aufruf zweier Methoden eine bestimmte Menge Zeit vergehen muss.
Die Konfiguration geschieht durch folgende Parameter:

- Der `clock`-Parameter referenziert eine Uhr.
- Der `action`-Parameter gibt an, welche Aktion ausgeführt wird. Der Uhr kann der Wert einer Konstanten oder einer anderen Uhr zugewiesen werden (`set`). Sie kann aber auch mit dem Wert einer Konstanten oder Uhr verglichen werden (`equals`, `smaller`, `equalsOrSmaller`, `bigger` und `equalsOrBigger`).
- Der `value`-Parameter beinhaltet den neuen Wert oder den Vergleichswert der Uhr. Beginnt der Wert mit einer Zahl, wird er als Konstante interpretiert - andernfalls als Bezeichner einer Uhr.

`TimeSet` ist eine Ereignismenge, die ausschließlich auf den Uhren des Jassda-Frameworks operiert. Die Dummy-Ereignisse, die von Jassda-Framework erzeugt werden, um festzustellen, ob Module an Ereignissen spezieller Methoden interessiert sind, enthalten keine Uhren-Informationen. Alle Dummy-Ereignisse werden von allen `TimeSet`s akzeptiert. Aus diesem Grund sollten `TimeSet`-Ereignismengen möglichst in Verbindung mit einer nicht `TimeSet` Ereignismenge benutzt werden.

Beispiel: Die folgende Ereignismenge stellt die Uhr `clock1` auf eine Sekunde (1000 Millisekunden).

```
{handler="TimeSet",
 clock="clock1",
 action="set",
 value="1000ms"}
```

Eine Ereignismenge, welche prüft, ob der Wert der Uhr `clock2` kleiner oder gleich dem Wert der Uhr `clock5` ist, wird definiert durch:

```
{handler="TimeSet",
 clock="clock2",
 action="equalsOrSmaller",
 value="clock5"}
```

□

4.5.4.3 Erstellen eigener Ereignismengen

Sollten die Implementierungen der von Jassda bereitgestellten Ereignismengen nicht ausreichen, können weitere Mengen implementiert werden. Neue Ereignismengen müssen die Java Klasse `JdiEventSet` erweitern und folgende Methoden implementieren:

contains Die Methode `contains` gibt an, ob ein konkretes Ereignis (`event`) zur Ereignismenge gehört oder nicht. Eine theoretische Betrachtung ist in Abschnitt 3.2.2.2 zu finden.

optimize Die Methode `optimize` vereinfacht komplexe Ereignismengen, die durch Anwendung des Schnitt- und Vereinigung-Operators entstehen können. Dabei werden Äquivalenzen aus Abschnitt 3.2.2.2 genutzt.

4.5.5 Variablen

Variablen sind Ereignismengen, denen zur Laufzeit ein Wert zugeordnet werden kann. Sie dienen dazu, Informationen aus konkreten Ereignissen für die weitere Verarbeitung zu speichern.

4.5.5.1 Abstraktion von Ereignissen

Nicht das konkrete Ereignis wird in der Variablen abgelegt, sondern eine Abstraktion. Bei der Zuweisung eines Wertes an eine Variable wird genau angegeben, welche Informationen des zu verarbeitenden Ereignisses in der Variablen gespeichert werden sollen. Alle anderen Informationen gehen verloren (von ihnen wird abstrahiert). Die Extraktion der gewünschten Daten aus dem Ereignis und Erstellung der entsprechenden Ereignismenge geschieht durch spezielle Java Klassen, die das Java Interface `jass.csp.Map` implementieren. Wie bei der Umsetzung der Ereignismengen bietet Jassda eine Implementierung, die in den häufigsten Fällen ausreichen sollte. Werden andere Abstraktionen vom Ereignis benötigt, können weitere Klassen implementiert werden.

4.5.5.2 Syntax

Die Wertzuweisung einer Variablen (intern als *EventSetInput* bezeichnet) wird durch folgende Syntax beschrieben:

EventSetInput ::= *EventSet* `"?"`
VariableIdentifier `":"` *MapDeclaration*

Dabei ist *EventSet* eine Ereignismenge und *VariableIdentifier* der Bezeichner der Variablen. Der Ausdruck *MapDeclaration* beschreibt die Klasse, welche Informationen des Ereignisses extrahiert und daraus ein neues Ereignis generiert:

MapDeclaration ::= `"["`
FieldDeclaration
(`","` *FieldDeclaration*)$*$
`"]"`

FieldDeclaration definiert das Format, wie Parameter für die Definition und Konfiguration der Klasse gesetzt werden.

FieldDeclaration ::= *Parameter*[`"["` *FieldIdentifier* `"]"`]
[`"="` *Field*[`"["` *FieldIdentifier* `"]"`]]

Die Konfiguration der Klasse, die die abstrahierende Abbildung implementiert, wird durch Attribut-Paare konfiguriert. Die Bedeutung der Paare wird durch die implementierende Klasse definiert.

Wie bei der Definition der Ereignismengen ist der Parameter `handler` für die Definition der implementierenden Klasse reserviert. In der Konfigurationsdatei des Trace-Prüfers kann eine Default-Klasse angegeben werden, die ausgewählt wird, wenn kein `handler` angegeben wird. Aliase können definiert werde, um Schreibarbeit bei der Angabe des Namens der Klasse zu sparen.

4.5.5.3 Standard-Abstraktion

Die Klasse `jass.debugger.jdi.eventset.GenericMap` implementiert die Abstraktion von Ereignissen. Die Abbildung wird durch Attribut-Paare konfiguriert. Ein Attribut bezeichnet dabei jeweils eine Eigenschaft des Ereignisses und kann optional durch eine Extension in eckigen Klammern konkretisiert werden. Die Paare beschreiben, welche Eigenschaften des Ereignisses auf welche Eigenschaft der Ereignismenge abgebildet werden.

Beispiel: Sei `a` eine Ereignismenge und `x` der Bezeichner einer Variablen.

1. Soll in der Variablen `x` die Information `method` gespeichert werden, kann folgende Wertzuweisung eingesetzt werden:

   ```
   a?x:[handler="jass.debugger.jdi.eventset.GenericMap",
        method=method]
   ```

   Die Klasse `jass.debugger.jdi.eventset.GenericMap` übernimmt die Abbildung des Ereignisses auf eine Ereignismenge. Sie wird konfiguriert durch das Attribut-Paar `method=method`. Der Wert des Attributs `method` wird aus dem Ereignis gelesen und als Attribut `method` in eine generierte Ereignismenge eingetragen. Die neue Ereignismenge wird der Variablen `x` zugeordnet. Sie akzeptiert alle Ereignisse, die von derselben Methode stammen, wie das Ereignis, welches abstrahiert wurde.

2. Falls die Klasse `jass.debugger.jdi.eventset.GenericMap` in der Konfugurationsdatei als Default-Klasse für die Abbildung eingetragen ist, kann die obige Wertzuweisung auch kürzer geschrieben werden:

   ```
   a?x:[method]
   ```

3. Die Attribute werden optional durch eine Erweiterung hinter dem Namen des Attributs konkretisiert.

   ```
   [arg[side]=result]
   ```

   Dies besagt, dass der Rückgabewert aus dem Ereignis als Methoden-Parameter mit dem Namen `side` abgelegt wird. Durch diese Konstruktion kann zum Beispiel geprüft werden, ob der Rückgabewert einer Methode später als Argument einer anderen Methode genutzt weird.

□

4.5.5.4 Erstellung eigener Abbildungen

Sollte die durch `GenericMap` implementierte Abbildung nicht den Anforderungen genügen, kann eine weitere Klasse definiert werden. Diese muss mindestens das Java Interface `jass.csp.Map` implementieren, welche folgende Methode deklariert:

getEventSet Diese Methode generiert eine Ereignismenge aus dem gegebenen Ereignis. Die aus dem Ereignis zu extrahierenden Information werden durch die Konfiguration der Klasse bestimmt.

configure Für jedes Attribut-Paar der Spezifikation wird die `configure`-Methode aufgerufen.

4.5.6 Prozesse

Der Trace-Prüfer akzeptiert Prozesse in der folgenden Syntax:

Process ::= *BasicProcess*
| *PrefixProcess*
| *SequenceProcess*
| *ChoiceProcess*
| *ParallelProcess*
| *QuantifiedProcess*
| *ReferenceProcess*

Die Prozesse der CSP_{jassda} -Spezifikation werden intern durch Klassen repräsentiert, die das Java Interface `jass.csp.CsProcess` implementieren. Dieses deklariert folgende öffentliche Methoden:

accept Die Methode `accept` entscheidet, ob ein Ereignis vom Prozess akzeptiert wird. Kann der Prozess das Ereignis verarbeiten (das heißt: es existiert laut der *operationellen Sematik* eine Transition, die den Prozess mit dem aktuellen Ereignis in einen Folge-Prozess überführt), liefert er den Folge-Prozess zurück. Andernfalls wird durch Werfen einer `NotAcceptedException` angezeigt, dass eine Verletzung der Trace-Zusicherung besteht.

calculateAlphabet Mit der Methode `calculateAlphabet` wird das Alphabet eines Prozesses berechnet. Die Berechnung folgt den in Kapitel 3.2.2.4 definierten Regeln.

newInstance Der CSP_{jassda} -Parser baut einen statische Struktur der in CSP_{jassda} -Spezifikation definierten Prozesse auf. Mit `newInstance` wird eine neue Prozess-Instanz erzeugt (eine Kopie der Prozess-Vorlage wird angelegt und initialisiert). Auf diese Weise kann eine Prozess-Definition gleichzeitig mehrfach genutzt werden.

optimize Bei der Verarbeitung von Ereignissen können sehr komplexe Prozesse entstehen. Die `optimize` Methode versucht den Prozess zu vereinfachen,

ohne dabei die Menge der beschriebenen Traces zu verändern. Die Optimierung wird auf Grundlage von Äquivalenzen in der Trace-Semantik (siehe Kapitel 3.2.4) durchgeführt.[3]

Jeder CSP_{jassda} -Prozess aus Kapitel 3.2 ist durch eine Java-Klasse im Paket `jass.csp` implementiert. Die Funktionalität ist direkt aus der theoretischen Betrachtung abgeleitet.

4.5.6.1 Einfache Prozesse

BasicProcess ::= `"STOP"`
| `"TERM"`
| *AnyProcess*

AnyProcess ::= `"ANY"`
| `"ANY" "["` *EventSet* `"]"`

Die einfachen Prozesse `STOP`, `TERM` und `ANY` werden von den Java-Klassen `jass.csp.Stop`, `jass.csp.Term` und `jass.csp.Any` auf Basis der operationellen Semantik und Trace-Semantik implementiert. Die Methode `calculateAlphabet` der Klasse `Stop` liefert die Ereignismenge `jass.csp.EmptySet`, welche kein Ereignis akzeptiert; `Term` antwortet mit der Ereignismenge `jass.csp.TermSet`, die nur das Terminierungs-Ereignis akzeptiert. Wurde in der Spezifikation des `ANY`-Prozesses ein Alphabet angegeben, wird dieses zurückgeliefert. Ansonsten wird dem `ANY`-Prozess das Alphabet des Prozesses zugeordnet, in dem er benutzt wird.

4.5.6.2 Prefix

PrefixProcess ::= *EventSet* `"->"` *Process*

Die Prefix-Operator wird durch die gleichnamige Klasse aus dem Paket `jass.csp` repräsentiert, welche mit zwei Argumenten (Ereignismenge und Folge-Prozess) instantiiert wird. Ist ein konkretes Ereignis in der gegebenen Ereignismenge enthalten, antwortet die `accept`-Methode mit dem Folge-Prozess. Andernfalls wird eine `NotAcceptedException` geworfen, um eine Vertragsverletzung anzuzeigen.

[3] Die Optimierung von Prozessen ist noch nicht implementiert

4.5.6.3 Sequence

SequenceProcess ::= *Process* `";"` *Process*

Die operationelle Semantik des Sequence-Operators (`P ; Q`) schreibt vor, dass der Prozess `P` so lange ausgeführt wird, bis `Q` das aktuelle Ereignis und `P` das Terminierungsereignis verarbeiten kann. Dieser Operator wird von der Klasse `jass.csp.Sequence` realisiert.

4.5.6.4 Auswahl

ChoiceProcess ::= *Process* `"[]"` *Process*

Jassda stellt mit der Klasse `jass.csp.Choice` eine Klasse für die Auswahl bereit. Sie implementiert die generalisierte Auswahl (siehe Kapitel 3.2.3.9). Die Anzahl der betrachteten Subprozesse ist nicht auf zwei festgelegt.

4.5.6.5 Parallel

ParallelProcess ::= *Process* `"||"` *Process*
| *Process* `"["` *Alphabet* `"||"` *Alphabet* `"]"` *Process*

Auch der Parallel-Operator ist in der generalisierten Form implementiert `jass.csp.Parallel`. Die Angabe der Alphabete der Subprozesse ist optional. Wenn keine Alphabete angegeben sind, werden sie aus den Subprozessen berechnet.

4.5.6.6 Quantifizierende Operatoren

Syntax Die quantifizierenden Operatoren werden durch folgende Syntax beschrieben:

QuantifiedProzess ::= *QuantifiedParallelProcess*
| *QuantifiedChoiceProcess*

QuantifiedParallelProcess ::= `"||"`
VariableIdentifier `":"`
MapDeclaration `"@"`
Process

QuantifiedChoiceProcess ::= `"[]"`
VariableIdentifier `":"`
MapDeclaration `"@"`
Process

Dabei ist *VariableIdentifier* der Bezeichner einer Variablen, *MapDeclaration* die Beschreibung der Abbildung und *Process* ein CSP_{jassda} -Process, der die Variable nutzen darf.

Theorie und Praxis Die disjunkte Zerlegung `M`, welche in Kapitel 3.2.2.2 eingeführt wurde, kann in der Praxis nicht explizit angegeben werden, da die Menge aller möglichen Ereignisse vor Beginn der Ausführung nicht bekannt ist.
Sind die Alphabete der Subprozesse des quantifizierenden Operators disjunkt, lässt sich die Mange `M` zur Laufzeit konstruieren, wie nachfolgend erläutert wird.

Algorithmus: Wie in Kapitel 3.2.2.2 beschrieben, wird die disjunkte Zerlegung durch eine Abbildung g definiert, welche die Menge der Ereignisse in disjunkte Ereignismengen abbildet:

$$\mathtt{M} = \bigcup_{\alpha \in Events} \{g(\alpha)\}$$

Wobei die Abbildung g wie folgt definiert ist.

$$\begin{aligned} g : Events &\to \mathbb{P}(Events) \\ \alpha &\mapsto g(\alpha) \\ g(\alpha) &= \{\beta \in \mathtt{a} \subseteq Events \mid \alpha \in \mathtt{a} \\ &\qquad \wedge\, \beta \text{ hat dieselbe Eigenschaft wie } \alpha\} \end{aligned}$$

Zur Laufzeit des Programmes kann die Menge `M` durch folgenden Algorithmus konstruiert werden:

1. Zu Beginn der Untersuchung ist die Menge `M` leer.
2. Ein zu untersuchende Ereignis α wird entsprechend der Abbildung g in eine Ereignismenge $g(\alpha)$ überführt.
3. Ist $g(\alpha)$ nicht in `M` enthalten, wird `M` um $g(\alpha)$ erweitert. Beim nächsten Ereignis α wird mit Punkt 2 fortgefahren.

Die quantifizierenden Operatoren (`[]x:M@P(x)` und `||x:M@P(x)`) verwenden die disjunkte Zerlegung `M`, um über bestimmte Eigenschaften zu quantifizieren.

Für jedes Element aus der Menge `M` verwalten sie einen Subprozess. Da `M` nicht bekannt ist, sind auch die zu betrachtenden Subprozesse nicht bekannt.
Analog der Konstruktion der Menge `M` wird auch die Menge der zu betrachtenden Subprozesse konstuiert: Bei jeder Erweiterung der Menge `M` um eine Ereignismenge $g(\alpha)$ wird ein weiterer Subprozess `P(`$g(\alpha)$`)` angelegt.
Diese Konstruktion funktioniert nur dann, wenn das Alphabet des parametrisierten Prozesses `P(`$g(\alpha)$`)` Teilmenge der Ereignismenge $g(\alpha)$ ist. Jedes Ereignis α kann also von maximal einem Subprozess verarbeitet werden.

4.5.6.7 Überwachter Aufruf

Der überwachte Aufruf wird von der Klasse `ProcessReference` ausgeführt. Diese sucht nach einem passenden Prozess in der Symboltabelle, instantiiert ihn und übergibt die Prozess-Parameter.

ReferenceProcess ::= *Label FormalParameters*

4.5.7 Überprüfen von Trace- und Zeit-Bedingungen

Die Trace-Semantik des CSP_{jassda} -Prozesses definiert die Menge der gültigen Traces.
Der Trace-Prüfer überprüft für jedes Ereignis, welches vom Jassda-Framework geliefert wird, ob der CSP_{jassda} -Prozess dieses gemäß der operationellen Semantik verarbeiten kann. Ist dies der Fall, wird der CSP_{jassda} -Prozess in den Folge-Prozess überführt, welcher sich aus der Semantik ergibt. Die Menge der gültigen Traces ist nun durch die Trace-Semantik des Folge-Prozesses definiert.

Eine Verletzung der Trace- und Zeit-Zusicherungen liegt vor, wenn ein Ereignis nicht verarbeitet werden kann. In diesem Fall kann der aktuelle CSP_{jassda} -Prozess in der grafischen Benutzungsoberfläche eingesehen werden. Zur Veranschaulichung der Darstellung des CSP_{jassda} -Prozesses zeigt Abbildung 4.6 die interne Darstellung der Spezifikation 4.5.2.

Das Ereignis, welches die Spezifikation verletzt hat, kann ebenfalls eingesehen werden. (Abbildung 4.7 zeigt zwei Threads. Das dargestellte Ereignis wurde vom Thread `helloThread` in der Methode `run()` ausgelöst.) Die Menge der gültigen Ereignisse, die durch den CSP_{jassda} -Prozess dargestellt wird, kann dem vom Jassda-Framework gelieferten Ereignis gegenübergestellt werden.

Spezifikation 4.5.2 Beispiel einer CSP_{jassda} -Spezifikation

```
trace trace1 {
   eventset helloThread     {class=".*.HelloThread"}
   eventset run             {method="run"}
   eventset init            {method="<init>"}

   hellothread() {
      helloThread.init.begin ->
      helloThread.init.end->
      helloThread.run.begin ->
      helloThread.run.end ->
      STOP
   }
}
```

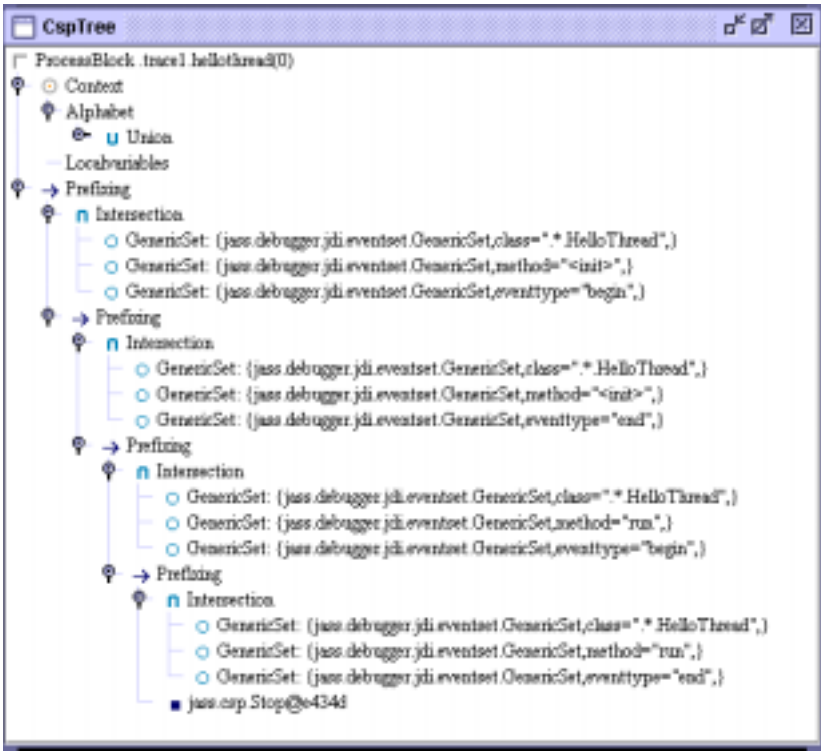

Abbildung 4.6: Ansicht eines CSP_{jassda} -Prozesses

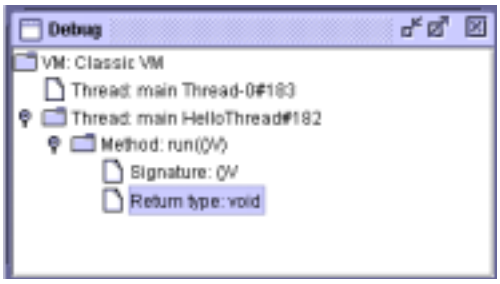

Abbildung 4.7: Ansicht eines Events

4.6 Ausgewählte Implementierungsdetails

4.6.1 Zugriff auf Rückgabewerte von Methoden

In Abschnitt 3.1.5.4 wurde bereits beschrieben, dass das Java Debug Interface keinen direkten Zugriff auf den Rückgabewert einer Methode gewährt. Um dennoch auf den Rückgabewert zuzugreifen, sind folgende Schritte notwendig:

1. Der Rückgabewert wird in eine lokale Variable geschrieben. Die Variable muss lokal sein, da bei Klassen- und Instanzvariablen Probleme bei paralleler Verarbeitung auftreten können.

2. Vor dem Beenden der Methode wird die lokale Variable mit Hilfe des JDI ausgelesen.

Nachfolgend werden die Möglichkeiten zur Realisierung dieser Schritte diskutiert und der von Jassda eingesetze Weg beschrieben.

Schreiben des Rückgabewertes in eine lokale Variable

Es bestehen mehrere Möglichkeiten, wie sichergestellt werden kann, dass der Rückgabewert in eine lokale Variable geschrieben wird:

1. Der *Programmierer* muss seinen Quelltext so gestalten, dass der Rückgabewert vor dem Beenden der Methode in einer bestimmten lokalen Variablen abgelegt ist. Dieser Ansatz erfordert viel Disziplin vom Software Entwickler und ist fehleranfällig. Bestehender Code müsste aufwändig aufbereitet werden. Java-Klassen, deren Quelltexte nicht verfügbar sind, lassen sich nicht anpassen.

2. Der *Quelltext* des Programmes kann mit einem Prä-Compiler bearbeitet werden, welcher das Programm *automatisch* nach `return`-Ausdrücken un-

tersucht und den Quelltext modifiziert. Dies hat wiederum den Nachteil, dass der Quelltext verfügbar sein muss.

3. Der *Bytecode* des Programmes wird *automatisch* modifiziert. Aufgrund der einfachen Struktur des Bytecodes sind die Rücksprünge aus den Methoden leicht zu lokalisieren. Der Rückgabewert liegt bei Verlassen der Methode auf dem Operand-Stack. Der Bytecode von Methoden mit Rückgabewert wird derart instrumentiert, dass vor Verlassen der Methode, der oberste Wert des Operand-Stack in eine neue lokale Variable mit einem speziellen Namen geschrieben wird. Um Namenskonflikt mit anderen Variablen zu vermieden, wird die neue Variable mit einem Namen bezeichnet der im Bytecode gültig, im Quellcode jedoch ungültig ist (zum Beispiel $result$). Dieser Ansatz eignet sich auch für Programme und Klassen-Bibliotheken, für die kein Quelltext verfügbar ist. Die einfache Struktur des Bytecodes vereinfacht die Analyse und Instrumentierung.

Im Jassda-Framework wird die Modifikation des Bytecodes des zu untersuchenden Programmes mit Hilfe der *Byte Code Engineering Library* (BCEL) (Dahm 2002) durchgeführt. Die Instrumentierung kann sowohl vor als auch während der Ausführung geschehen:

- Ein Post-Compiler modifiziert die Class-Dateien des Programmes *vor* dessen Ausführung.
- Zum Laden der Klassen des untersuchten Programmes wird ein spezieller Class Loader eingesetzt. Dieser liest die originalen Class-Dateien aus dem Dateisystem und reicht eine modifizierte Version an die Virtual Machine weiter.

Abfrage der lokalen Variable vor dem Beenden der Methode

Der Debuggee kann über das JDI das Ende einer Methode durch folgende Ereignisse anzeigen:

MethodExitEvent Das JDI bietet mit dem `MethodExitEvent` ein spezielles Ereignis welches das Ende einer Methode anzeigt. Dieses Ereignis wird *nach* dem Beenden der Methode erzeugt. Zu diesem Zeitpunkt ist die lokale Variable nicht mehr gültig. Außerdem stellt das `MethodExitEvent` keine Methode für den Zugriff auf den Rückgabewert bereit.

BreakpointEvent Durch setzen eines Breakpoints vor den Rücksprüngen aus der Methode wird ein `BreakpointEvent` *vor* dem Beenden der Methode erzeugt. Um die richtigen Positionen für die Breakpoints zu finden, muss der Bytecode der Methode untersucht werden.

Der Bytecode-Analyzer des Jassda-Frameworks analysiert den Bytecode der Methode und setzt vor jeden Rücksprung-Opcode einen Breakpoint.

4.6.2 Erstellen von Debug-Informationen in Class-Dateien

Um die volle Funktionalität des JDI nutzen zu können, müssen die betrachteten Java Klassen Debug Informationen enthalten. Dies ist in der Regel nur dann der Fall, wenn es bei der Übersetzung explizit angegeben wurde. Die *Byte Code Engineering Library* kann bestehende Class-Dateien um generische Debug-Informationen erweitern.

4.6.3 Logging

Die Ausführungsschritte von Jassda werden protokolliert. Dazu nutzt Jassda einen Logging-Mechanismus auf Basis der frei verfügbaren Bibliothek *log4j*. Einzelnen Komponenten lassen sich getrennte Log-Dateien zuordnen. An zentraler Stelle kann die Logging-Ebene konfiguriert werden, welche die Detailliertheit der Log-Ausgaben steuert. Die Logging-Ebenen *error*, *warn*, *info* und *debug* stehen zur Auswahl. Wird die Log-Ebene auf *error* gestellt, schreibt Jassda nur kritische Fehler in die Log-Dateien. Wird andererseits die Log-Ebene auf *debug* eingestellt, werden sehr viele Informationen protokolliert - eingeschlossen die Informationen, die in den Log-Ebenen *error*, *warn* und *info* ausgegeben werden.

4.6.4 Zahlen zur Größe der Implementierung

Die Implementierung von Jassda wurde in die Klassen-Hierarchie des Jass-Tools integriert. Folgende Pakete sind angelegt worden.

4.6.4.1 Das Paket `jass.csp`

Das Paket `jass.csp` enthält sämtliche Klassen, welche für die Verarbeitung von CSP_{jassda} -Operatoren notwendig sind. Bei der Umsetzung wurde darauf geachtet, dass dieses Paket unabhängig vom restlichen Jassda-Tool einsetzbar ist. Das Paket enthält 54 Klassen und insgesamt 2600 Zeilen Code.

4.6.4.2 Das Paket `jass.debugger`

Das `jass.debugger`-Paket nimmt die Klassen des Jassda-Frameworks und der Module auf. Zur besseren Übersicht sind die Klassen in folgenden Unterpaketen angeordnet:

jass.debugger.bytecode Die Klassen, die Bytecode manipulieren und analysieren sind hier untergebracht. (5 Klassen, 470 Zeilen Code)

jass.debugger.examples Einige Beispiel-Programme können zum Testen der Funktionalität von Jassda eingesetzt werden. In diesem Paket sind auch die Klassen untergebracht, die in der Fallstudie eingesetzt wurden. (10 Klassen, 1050 Zeilen Code)

jass.debugger.gui Die Klassen, welche die grafischen Benutzungsoberfläche das Jassda-Frameworks realisieren, befinden sich im GUI-Paket. (24 Klassen, 1200 Zeilen Code)

jass.debugger.jdi Das Java Debug Interface wird durch die Klassen des jdi-Pakets gekapselt. (36 Klassen, 2600 Zeilen Code)

jass.debugger.modules Im Modules-Paket sind die Trace-Checker und Logger Module untergebracht. Die Implementierung des Trace-Prüfers beansprucht dabei den meisten Raum. (86 Klassen, 15900 Zeilen Code)

Kapitel 5

Fallstudie

Im Rahmen dieser Fallstudie wird die Anwendbarkeit von Jassda in den Bereichen

1. Anwendungen mit mehreren Threads
2. Applets
3. Servlets
4. verteilte Systeme

gezeigt. Außerdem wird der Einfluss von Jassda auf das Laufzeitverhalten der untersuchten Anwendung untersucht.

5.1 Einsatzgebiete von Jassda

Jassda wurde mit folgenden Szenarien getestet:

HelloWorld: Ein kleines HelloWorld-Programm schreibt mit mehreren Threads die Worte ”Hello World” auf die Konsole. Mit diesem Szenario wird die Anwendbarkeit von Jassda für Programme mit mehreren Threads getestet.

Verteilte Anwendung mit Applets: Im Java Tutorial von Campione, Walrath, und Huml (2000b) wird die Implementierung eines verteilten Systems gezeigt. Es beschreibt, wie zwei Applets über einen zentralen Server miteinander kommunizieren. Jassda kann gleichzeitig mit den Applets und dem Server verbunden werden und die Kommunikation zwischen ihnen beobachten und prüfen.

Servlets: Eine kleine Servlet-Anwendung, die einen Namen vom Benutzer erfragt und diesen in der darauf folgenden HTML-Seite ausgibt, zeigt, wie auch Servlets untersucht werden können.

Die Anwendung von Jassda auf die Szenarien zeigt, dass Jassda in allen Bereichen eingesetzt werden kann. Der Leser kann die Anwendung von Jassda anhand der Beispiele und Konfigurations-Dateien im Quellpaket von Jassda nachvollziehen.

5.2 Benchmark

Die Extraktion einer Trace als auch ihre Auswertung beanspruchen Zeit. Dieser Overhead wird mit Hilfe eines kleinen Benchmark-Programmes ermittelt. Die vorliegenden Ergebnisse basieren auf einer einfachen Implementierung des Bubblesort-Sortierverfahrens. Der Bubblesort-Algoritmus wurde ausgewählt, weil die Umsetzung sehr einfach ist und er sich leicht umformulieren lässt. Der Quelltext der verwendeten Implementierung ist in Anhang A aufgelistet.
Der Bubblesort-Benchmark sortiert eine Liste von 10000 Zahlen und misst jeweils die Zeiten, welche die Sortier-Methode (`sort1` und `sort2`) dafür benötigen. Diese Methoden unterscheiden sich lediglich dadurch, dass die innere Schleife des Algorithmus bei `sort2` in eine eigene Methode `exchange` ausgelagert ist. Jassda wird so konfiguriert, dass alle Methoden der Klasse Bubblesort beobachtet werden. Beim Aufruf der Sortier-Methode `sort1` werden zwei Ereignisse erzeugt (Beginn und Ende der Methode `sort1`). Der Aufruf von `sort2` erzeugt aufgrund der ausgelagerten inneren Schleife 20002 Ereignisse (10000-facher Aufruf der `exchange`-Methode). Als Testumgebung dient ein Pentium III mit 900MHz und dem Betriebssystem Microsoft Windows 2000. Es wurde die Java Virtual Machine des Sun JDK 1.3.1 eingesetzt.
Die Ergebnisse werden in der Tabelle 5.1 zusammengefasst. Dort sind die Messergebnisse für unterschiedliche Konfigurationen der Hotspot Java Virtual Machine und der Classic Java Virtual Machine aufgeführt. Beide Virtual Machines sind im JDK enhalten und lassen sich durch die Parameter `-hotspot` beziehungsweise `-classic` beim Start der Virtual Machine angeben. Die Hotspot VM bietet einen ”Just in Time”-Compiler (JIT) sowie weitere Optimierungen. Die Classic VM muss den Bytecode vollständig interpretieren. Die Ziffern 1 und 2 symbolisieren die Methoden `sort1` und `sort2` der Bubblesort-Implementierung (siehe Anhang A).
Die ersten vier Zeilen zeigen, ob sich die Optimierung des Bytecodes (Übersetzung mit Java-Compiler-Option ”-O”) oder die Integration von Debug-Informationen (Übersetzung mit Java-Compiler-Option ”-g”) in der Ausführungsgeschwindigkeit niederschlägt. Anschließend wird der Benchmark in einer Virtual

Beschreibung	hotspot		classic	
	1	2	1	2
optimierter Code ohne Debug-Informationen	1,302*s*	1,262*s*	11,897*s*	12,067*s*
optimierter Code mit Debug-Informationen	1,332*s*	1,252*s*	11,897*s*	12,067*s*
nicht optimierter Code ohne Debug-Informationen	1,302*s*	1,252*s*	11,907*s*	12,067*s*
nicht optimierter Code mit Debug-Informationen	1,342*s*	1,252*s*	11,918*s*	12,057*s*
VM im Debug-Modus	13,629*s*	13,630*s*	50,983*s*	51,013*s*
VM im Profiling-Modus	1,442*s*	1,372*s*	12,458*s*	12,458*s*
VM mit Jassda verbunden	15,031*s*	2310,393*s*	56,812*s*	629,095*s*

Tabelle 5.1: Ergebnisse des Benchmarks

Machine ausgeführt, die im Debug-Modus gestartet wurde. Zum Vergleich wird auch der Profiling-Modus aufgeführt. Dieser Modus ist speziell für die Untersuchung des Laufzeitverhaltens der Virtual Machine (VM) entwickelt. Im Anschluss wird Jassda mit einer im Debug-Modus gestarteten VM verbunden.

Aus den Messergebnissen lässt folgendes ableiten:

- Der Bytecode der Bubblesort-Implementierung lässt sich kaum von Java Compiler optimieren.
- Die Ausführungsgeschwindigkeit ändert sich durch Debug-Informationen wenig.
- Die Hotspot Virtual Machine, welche einen ”Just in Time”-Compiler einsetzt, arbeitet zehn mal schneller als die Classic Virtual Machine.
- Wird die Virtual Machine im Debug-Modus gestartet, verlangsamt sich die Ausführungsgeschwindigkeit der Hotspot Virtual Machine um den Faktor zehn. Die Classic Virtual Machine benötigt die fünf-fache Zeit.
- Im Profiling-Modus verringert sich die Geschwindigkeit unwesentlich.
- Die Ausführungsgeschwindigkeit von Code-Fragmenten, die Jassda nicht untersucht, unterscheidet sich kaum von der Geschwindigkeit im Debug-Modus.

- Programm-Fragmente, für die Jassda zur Kontrolle der Traces Breakpoints eingefügt hat werden deutlich langsamer ausgeführt. Hier fällt auf, das die Hotspot Virtual Machine sehr lange benötigt, um Breakpoints zu verarbeiten.

Zusammenfassend lässt sich sagen, dass die Ausführungsgeschwindigkeit eines Programmes, welches von Jassda untersucht wird, um den Faktor 10 (wenn sehr wenige Breakpoints gesetzt werden müssen) und 2000 (wenn sehr viele Breakpoints gesetzt werden) verringern kann.

5.3 Diskussion: Messung von Zeit beim Debugging

Der Ergebnisse des Benchmarks zeigen deutlich, dass sich die Laufzeiteigenschaften eines Programmes stark durch den Einsatz von Jassda verändern können. Je mehr Ereignisse des untersuchten Programmes ausgewertet werden müssen, desto geringer wird die Geschwindigkeit. Die Definition von zeitlichen Bedingungen für ein Programm dessen Ausführungsgeschwindigkeit sich durch den Einsatz von Jassda um einen Faktor von 2000 verringern kann, erscheint nur in sehr wenigen Fällen sinnvoll.
Ein weiteres Problem bei der Spezifikation von Zeit in Java-Programmen ist die schlechte Realzeitfähigkeit. Desweiteren ist den Genauigkeit der Systemzeit häufig nicht ausreichend. Diese Eigenschaft wirkt sich auf die Auswertung der Laufzeit-Uhren von Jassda aus. Sehr kurze Zeiträume lassen sich nicht erfassen. Wird Jassda in der Standard Virtual Machine des JDK 1.3 unter Microsoft Windows 2000 gestartet, lassen sich Zeiträume, die kleine als 5 Millisekunden sind, nicht erfassen. Es wäre zu prüfen, ob realzeitfähige Implementierungen der Java Virtual Machine diese Einschränkung beseitigen können.

Kapitel 6

Ergebnisse

6.1 Einordnung

Im Rahmen dieser Arbeit wurde die Software-Architektur Jassda entwickelt, die zur Ermittlung und Validieren von Traces zur Laufzeit von Java-Programmen eingesetzt werden kann. Die Implementierung bietet ein konfigurierbares Framework, welches die Informationen über den Programm-Ablauf unter Verwendung des Java Debug Interfaces (JDI) extrahiert. Diese Informationen können von verschiedenen Modulen ausgewertet werden. Ein Logging-Modul, welches den Ablauf des untersuchten Programmes in eine Datei protokolliert, sowie ein Trace-Prüfer-Modul, welches eine Trace zur Laufzeit validiert, sind implementiert. Weitere Module können beliebige weitere Funktionalitäten realisieren.
Der Entwicklung des Frameworks ist eine Analyse aktueller Tools und Ansätze zur Integration des Konzeptes *Programmieren mit Vertrag* in die Programmiersprache vorausgegangen. Das Konzept des *Programmieren mit Vertrag* ist bereits mehrfach für die Programmiersprache Java umgesetzt, wie das Kapitel 2 zeigt. Abgesehen von einem Forschungsprojekt von Lucent Technologies (heute: Agere Systems) existieren aber keine Produkte, die Java-Programme unter Verwendung des Java Debug Interfaces validieren.
Das Trace-Prüfer-Modul nutzt den, in dieser Arbeit entwickelten, CSP-Dialekt CSP_{jassda} als Spezifikationssprache. Zur Laufzeit prüft es, ob eine Trace eines Java-Programmes in der Menge gültiger Traces enthalten ist, welche mit CSP_{jassda} spezifiziert wurde.

6.2 Fazit

In der Fallstudie wurde gezeigt, dass das Tool Jassda für unterschiedliche Java Applikationen einsetzbar ist. Es können Programme mit mehreren Threads,

Servlets, Applets als auch verteilte Systeme untersucht werden. Die Messergebnisse des Benchmarks zeigen aber auch die eingeschränkte Anwendbarkeit der Zeit-Zusicherungen.
Jassda ermöglicht den Einsatz formaler Methoden zur Prüfung von Traces zur Laufzeit. Zur Validierung von Zeit-Zusicherungen ist aufgrund des hohen Einflusses der Prüfung auf das Laufzeitverhalten des untersuchten Programmes weniger geeignet.

6.3 Ausblick

Die *Java with Assertions Debugger Architecture* ist im Rahmen dieser Arbeit prototypisch implementiert, Erweiterungsmöglichkeiten werden nachfolgend beschrieben.

Integration in Entwicklungsumgebungen

Das Tool Jassda ist mit einer grafischen Benutzungsoberfläche ausgestattet. Diese erlaubt, die grundlegende Funktionalität zu nutzen. Wesentlich komfortabler hingegen wäre die Integration in eine grafische Entwicklungsumgebung.
Im Rahmen des *NetBeans*-Projektes (Netbeans 2002) wird eine Entwicklungsumgebung für Java entwickelt. Diese ermöglicht die Integration verschiedener Komponenten. Jassda könnte somit einerseits das *NetNeans*-Projekt erweitern und andererseits Software-Komponenten aus dem Projekt einsetzen, die beispielsweise komfortable Editoren realisieren.

Erstellen weiterer Module

Die *Java with Assertions Debugger Architecture* kann als Basis für weitere Tools eingesetzt werden, die Informationen über die Ausführung eines Java-Programmes benötigen:

1. Durch Implementierung eines Moduls zur Überprüfung von Vor- und Nachbedingungen von Methoden ließe sich das klassische Konzept des *Programmierens mit Vertrag* umsetzen.

2. Das übliche Vorgehensmodell bei der Software-Entwicklung sieht nach die Implementierung eine Test-Phase vor. Die Wahl der Testdaten, die einen möglichst großen Teil der des Codes abdecken, ist schwierig. Ein Jassda-Modul könnte Statistiken über die Qualität der Testdaten erzeugen und somit Aufschluss über die Programm-Abdeckung liefern. Ein weiterer Ansatz

könnte auch das automatische Generieren von Testdaten aus dem internen Ablauf des untersuchten Programmes darstellen. Dies ist insbesondere für interaktive Anwendungen interessant.

Optimierung der Ausführungsgeschwindigkeit

Die Ausführungsgeschwindigkeit einer Java-Applikation, welche im Debug-Modus gestartet ist, sinkt um den Faktor fünf bis zehn. Werden darüber hinaus Breakpoints gesetzt, verschlechtert sich die Geschwindigkeit weiter. Es wäre interessant, welche Möglichkeiten der Optimierung vorhanden sind. Beispielsweise könnte das *Java Virtual Machine Profiling Interface* auf die Anwendbarkeit im Bereich der Trace- und Zeit-Zusicherungen evaluiert werden.

Anhang A

Implementierung des Benchmarks

Der Benchmark zur reinen Performancemessung basiert auf einer einfachen Implementierung des Bubblesort-Sortierverfahrens. Eine Liste von 10000 Zahlen wird mit zwei unterschiedlichen Implementierungen des Bubblesorts sortiert:

1. Die Methode `sort1` sortiert die Liste indem eine zwei-fach geschachtelte Schleife eingesetzt wird.

2. Der zweite Ansatz (`sort2`) lagert die innere Schleife in eine weitere Methode aus. Auf diese Weise kann die Zeit für Methodenaufrufe unter verschiedenen Bedingungen gemessen werden.

Nachfolgend wird der Quelltext der Bubblesort-Implementierung dargestellt.

```
package jass.debugger.examples.benchmark;

public class Bubblesort {
  private static int[] field1;
  private static int[] field2;
  private static int count = 10000;

  private static void fillField(int[] field)
  {
    int zufall = 1235541;
    for(int i = 0; i < field.length;
        zufall = (zufall * 231234141 + 1) % ((1<<31)-1))
      field[i++] = zufall;
  }

  private static int[] copyField(int[] feld) {
    int[] neuesFeld = new int[feld.length];
    for (int i = 0; i < feld.length ; i++) {
      neuesFeld[i] = feld[i];
    }
```

```
  return neuesFeld;
}


private static void sort1(int[] feld)
{
  for(int element = 0; element < feld.length; element++)
    for(int lauf = feld.length; --lauf > element; )
      if(feld[lauf-1] > feld[lauf])
      {
        int tausch = feld[lauf-1];
        feld[lauf-1] = feld[lauf];
        feld[lauf] = tausch;
      }
}

private static void sort2(int[] field)
{
  for(int element = 0; element < field.length; element++) {
      exchange(field,element);
  }
}

private static void exchange(int[] feld, int element) {
  for(int lauf = feld.length; --lauf > element; )
    if(feld[lauf-1] > feld[lauf])
    {
      int tausch = feld[lauf-1];
      feld[lauf-1] = feld[lauf];
      feld[lauf] = tausch;
    }
}


private static void printField(int[] field)
{
  for(int i = 0; i < field.length; ) {
    System.out.print(field[i++] + " ");
  }
  System.out.println();
}

public static void main(String[] args)
{
  long starttime1, stoptime1;
  long starttime2, stoptime2;

  System.out.println("Benchmark: Bubblesort with " + count + " numbers");
```

```
    // field1 = field2
    field1 = new int[count];
    field2 = new int[count];
    fillField(field1);
    field2 = copyField(field1);

    // sort field1
    starttime1 = System.currentTimeMillis();
    sort1(field1);
    stoptime1 = System.currentTimeMillis();

    // sort field2
    starttime2 = System.currentTimeMillis();
    sort2(field2);
    stoptime2 = System.currentTimeMillis();

    // print results
    System.out.println("Sort-time1: " + (stoptime1 - starttime1) + " ms.");
    System.out.println("Sort-time2: " + (stoptime2 - starttime2) + " ms.");
  }
}
```

Literatur

AADEBUG (2000). Fourth International Workshop on Automated Debugging. URL: http://www.irisa.fr/lande/ducasse/aadebug2000.

Aho, Alfred V.; Sethi, Ravi und Ullman, Jeffrey D. (1988). *Compilerbau*. Addison-Wesley.

AOSD steering committee (2001). Aspect Oriented Software Development. URL: http://aosd.net/.

AspectJ Homepage (2001). AspectJ: crosscutting objects for better modularity. URL: http://www.aspectj.org.

Bakkers, Andy (2001). Communicating Threads for Java: Real-Time Extension for Java. URL: http://www.rt.el.utwente.nl/javapp/.

Bartetzko, Detlef (1999, 4). Parallelität und Vererbung beim Programmieren mit Vertrag. Diplomarbeit, Carl von Ossietzky Universität Oldenburg.

Bartetzko, Detlef; Fischer, Clemens; Möller, Michael und Wehrheim, Heike (2001). Jass - java with assertions. In K. Havelund und G. Rosu (Hrsg.), *Electronic Notes in Theoretical Computer Science*, Band 55. Elsevier Science Publishers. URL: http://www.elsevier.nl/gej-ng/31/29/23/83/33/28/55.2.002.ps.

Bennett, Laura (2001, 2). Java debugging. URL: http://www6.software.ibm.com/developerworks/education/j-debug/.

Bowen, Jonathan (2001). The CSP archive. URL: http://www.afm.sbu.ac.uk/csp/.

Bracha, Gilad; Gosling, James; Joy, Bill und Steele, Guy (2000). *The Java Language Specification, Second Edition*. Addison Wesley.

Campione, Mary; Walrath, Kathy und Huml, Alison (2000a, 1). *The Java Tutorial: A Short Course on the Basics*. Addison-Wesley. URL: http://java.sun.com/docs/books/tutorial/index.html.

Campione, Mary; Walrath, Kathy und Huml, Alison (2000b, 1). *The Java Tutorial: A Short Course on the Basics*, Kapitel: Using a Server to Work

Around Security Restrictions. Addison-Wesley. URL: http://java.sun.com/docs/books/tutorial/applet/practical/workaround.html.

Clarke, Edmund M.; Wing, Jeannette M.; Alur, Rajeev et al. (1996). Formal methods: state of the art and future directions. *ACM Computing Surveys 28*(4), 626–643. URL: http://www-2.cs.cmu.edu/afs/cs/usr/wing/www/mit/paper/paper.html.

Cleve, Holger und Zeller, Andreas (2001). Automatisches Debugging. Technischer Bericht, Lehrstuhl für Softwaresysteme, Universität Passau.

Dahm, Markus (2002). Homepage: Byte Code Engineering Library. URL: http://bcel.sourceforge.net/.

Duncan, Andrew und Hölzle, Urs (1998). Adding Contracts to Java with Handshake. Technischer Bericht, Department of Computer Science, University of California, Santa Barbara.

Enseling, Oliver (2001, 2). iContract: Design by Contract in Java. Technischer Bericht, Java World. URL: http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-cooltools.html.

Fischer, Clemens (2000). *Combination and Implementation of Processes and Data: From CSP-OZ to Java*. Doktorarbeit, Carl von Ossietzky Universität Oldenburg.

Fowler, Martin (1997). *Analysis Patterns: Reusable Object Models*, Kapitel: Design By Contract. Addison Wesley. URL: http://www.awl.com/cseng/titles/0-201-89542-0/techniques/designByContract.htm.

Freitas, Leonardo (2002). Jack. Diplomarbeit, Federal University of Pernambuco - UFPE, Center of Informatics Recife, PE, Brazil.

Hoare, C.A.R. (1978, 8). Communicating Sequential Processes. *Communications of the ACM*.

Hoare, C.A.R. (1985). *Communicating Sequential Processes*. Prentice-Hall International.

Holub, Allen I. (2001). Inside The Java VM. URL: http://www.holub.com/class/java_vm/notes.html.

Jass (2001). The Jass Page. URL: http://semantik.informatik.uni-oldenburg.de/~jass/.

Java Developer Connection, Bug Database (1998, 12). Bug 4195445. JVMDI spec: Add return value to Method Exit Event. URL: http://developer.java.sun.com/developer/bugParade/bugs/4195445.html.

Java Developer Connection, Bug Database (2001, 9). Bug 4508774. JVMDI spec: Would like the function GetOperandStack in jvmdi to be implemented. URL: http://developer.java.sun.com/developer/bugParade/bugs/4508774.html.

Java-RT Resources (2001). URL: http://tao.doc.wustl.edu/rtj/.

JavaCC Homepage (2001). Java Compiler Compiler (JavaCC) - The Java Parser Generator. URL: http://www.webgain.com/products/java_cc/.

Karaorman, Murat; Hölzle, Urs und Bruno, John (1998). jContractor: A Reflective Java Library to Support Design By Contract. Technischer Bericht, Department of Computer Science, University of California, Santa Barbara. URL: http://www.cs.ucsb.edu/~murat/jContractor.PDF.

Kiczales, Gregor; Hilsdale, Erik; Hugunin, Jim; Kersten, Mik; Palm, Jeffrey und Griswold, William G. (2001). An Overview of AspectJ. In *ECOOP*, S. 327–353. URL: http://aspectj.org/documentation/papersAndSlides/ECOOP2001-Overview.pdf.

Kim, Moonjoo (2001). *Information Extraction for Run-time Formal Analysis*. Doktorarbeit, Computer and Information Science Department, University of Pennsylvania, in Vorbereitung.

Kim, Moonjoo; Kannan, Sampath; Lee, Insup; Sokolsky, Oleg und Viswanathan, Mahesh (2001). Java-MaC: a Run-time Assurance Tool for Java Programs. In K. Havelund und G. Rosu (Hrsg.), *Electronic Notes in Theoretical Computer Science*, Band 55. Elsevier Science Publishers. URL: http://www.elsevier.nl/gej-ng/31/29/23/83/33/35/55.2.009.ps.

Kramer, Reto (2001). iContract: The Java Design by Contract Tool. Homepage: http://www.reliable-systems.com/tools/iContract/iContract.htm.

Leavens, Gary (2001). The Java Modeling Language (JML). URL: http://www.cs.iastate.edu/~leavens/JML.html.

Leavens, Gary T.; Leino, K. Rustan M.; Poll, Erik; Ruby, Clyde und Jacobs, Bart (2000). JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, S. 105–106. URL: ftp://ftp.cs.iastate.edu/pub/techreports/TR00-15/TR.ps.gz.

Lencevicius, Raimondas (2000). On-the-fly Query-Based Debugging with Examples. In *AADEBUG 2000, Fourth International Workshop on Automated Debugging*. URL: http://www.irisa.fr/lande/ducasse/aadebug2000/proceedings/10lencevicius.ps.gz.

Lencevicius, Raimondas; Hölzle, Urs und Singh, Ambuj K. (1997, 10). Query-Based Debugging of Object-Oriented Programs. In *OOPSLA97, 12th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*. URL: `http://www.cs.ucsb.edu/labs/oocsb/papers/oopsla97.pdf`.

Lindholm, Tim und Yellin, Frank (1999). *The Java Virtual Machine Specification*. Addison Wesley. URL: `http://java.sun.com/docs/books/vmspec/index.html`.

Man Machine Systems (2000). Design by Contract for Java Using JMSAssert. URL: `http://www.mmsindia.com/DBCForJava.html`.

Meemken, Dieter (1997, 9). Programmieren mit Vertrag in Java. Diplomarbeit, Carl von Ossietzky Universität Oldenburg.

Meyer, Bertrand (1998). *Object-Oriented Software Construction*. Prentice Hall.

Meyer, Bertrand (2001). Building bug-free O-O software: An introduction to Design by Contract. URL: `http://www.eiffel.com/doc/manuals/technology/contract/`.

Murray, David J. und Parson, Dale E. (2000). Automated Debugging in Java Using OCL and JDI. URL: `http://www.irisa.fr/lande/ducasse/aadebug2000/proceedings/02murray.ps.gz`.

Museophile (2001). The CSP archive. URL: `http://www.afm.sbu.ac.uk/csp/`.

Netbeans (2002). Homepage: Netbeans. URL: `http://www.netbeans.org/`.

Olderog, Ernst-Rüdiger (2001). Spezifikation von Daten und Prozessen mit Z und CSP.

Plath, Michael (2000, 6). Trace-Zusicherungen in Jass, Erweiterung des Konzepts Programmieren mit Vertrag. Diplomarbeit, Carl von Ossietzky Universität Oldenburg.

Plotkin, G.D. (1981). A structural approach to operational semantics. Technischer Bericht, Department of Computer Science, Aarhus University.

Plotkin, G.D. (1982). An operational semantics of CSP. *Formal description of programming concepts*.

Pöschmann, Thomas (2001a, 11). Design by Contract mit Jcontract und iContract. *Java Magazin*.

Pöschmann, Thomas (2001b, 10). Vor- und Nachbedingungen in Java, Teil 1: Konzepte und Tools. *Java Magazin*.

Reilly, David (2000, 4). Debugging in Java: Debugging Theory and Strategies. URL: http://webdeveloper.earthweb.com/scripting/webjava/article/0,,12230_625961,00.html.

Robinson, Matthew und Vorobiev, Pavel (2000). *Swing*. Manning.

Roscoe, Bill (1997). *The Theory and Practice of Concurrency*. Prentice Hall International Series in Computer Science.

Roulo, Mark (1998, 9). Accelerate your Java apps!, Where does the time go? Find out with these speed benchmarks. *Java World*. URL: http://www.javaworld.com/javaworld/jw-09-1998/jw-09-speed.html.

Schneider, Steve (2000). *Concurrent and Real-time Systems, The CSP Approach*. John Wiley. URL: http://web.comlab.ox.ac.uk/oucl/publications/books/concurrency/.

Snyder, Carolyn (2001, 11). Paper Prototyping. URL: ftp://www6.software.ibm.com/software/developer/library/us-paper.pdf.

Sun Community Source (2001). Java 2 Platform, Standard Edition. URL: http://www.sun.com/software/communitysource/java2/index.html.

Sun Microsystems, Inc. (2001). Java Platform Debugger Architecture. URL: http://java.sun.com/j2se/1.4/docs/guide/jpda.

Venners, Bill (1997a, 1). How the Java virtual machine handles exceptions. *Java World*. URL: http://www.javaworld.com/javaworld/jw-01-1997/jw-01-hood.html.

Venners, Bill (1997b, 7). How the Java virtual machine handles method invocation and return. *Java World*. URL: http://www.javaworld.com/javaworld/jw-06-1997/jw-06-hood.html.

Venners, Bill (1999). *Inside The Java Virtual Machine*. Osborne McGraw-Hill. URL: http://artima.com/insidejvm/ed2/index.html.

Welch, Peter (2001). Communicating Sequential Processes for Java (JCSP). URL: http://www.cs.ukc.ac.uk/projects/ofa/jcsp/.

Zörner, Stefan (2001, 11). Die Assertion-Funktionalität des JDK1.4. *Java Magazin*.

Index

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe.

Oldenburg, 16. Juli 2002